



POLITECNICO DI TORINO

Mater degree course in Computer Engineering

Master Degree Thesis

# Quantum Key Distribution in softwarised networks

**Supervisors**

prof. Antonio Lioy

dott. Ignazio Pedone

**Candidate**

Chiara RUGGERI

DECEMBER 2020



# Contents

<b>1</b>	<b>Introduction</b>	6
<b>2</b>	<b>Quantum Computing</b>	7
2.1	Quantum Information and Quantum computing . . . . .	7
2.1.1	Qubit . . . . .	7
2.1.2	Bra-ket notation . . . . .	8
2.1.3	Superposition . . . . .	9
2.1.4	Entanglement . . . . .	10
2.1.5	Measurement . . . . .	11
2.2	Quantum computing . . . . .	12
2.2.1	Single-qubit gates . . . . .	13
2.2.2	Multi-qubit gates . . . . .	14
2.2.3	Quantum circuit . . . . .	16
2.3	From classical to quantum cryptography . . . . .	17
2.3.1	Theoretical secure communication . . . . .	17
2.3.2	Classical cryptography threats . . . . .	17
2.3.3	Post quantum cryptography . . . . .	18
2.3.4	Quantum Cryptography . . . . .	19
<b>3</b>	<b>Quantum Key Distribution</b>	22
3.1	QKD overview . . . . .	22
3.1.1	No-cloning principle . . . . .	22
3.1.2	Random Number Generators . . . . .	23
3.1.3	Quantum and classical channels . . . . .	24
3.2	First example with a noiseless channel . . . . .	25
3.3	Noise introduction and almost perfect security . . . . .	26
3.3.1	Error correction problem . . . . .	27

3.3.2	Privacy amplification . . . . .	28
3.4	Quantum Key Distribution Protocols . . . . .	28
3.4.1	BB84 . . . . .	29
3.4.2	E91 . . . . .	31
<b>4</b>	<b>QKD standard</b>	<b>33</b>
4.1	ETSI GS QKD . . . . .	33
4.1.1	Application Program Interface . . . . .	34
4.1.2	REST-based interface . . . . .	37
<b>5</b>	<b>QKD criticalities and known attacks</b>	<b>41</b>
5.1	Classical channel . . . . .	41
5.2	Repeaters . . . . .	42
5.3	Time shift attack . . . . .	43
5.4	Photon number splitting attack . . . . .	44
5.5	Man In The Middle (MITM) . . . . .	46
5.6	Other attacks . . . . .	46
<b>6</b>	<b>Qiskit framework</b>	<b>48</b>
6.1	Introduction . . . . .	48
6.1.1	Components . . . . .	48
6.1.2	How to simulate a quantum circuit . . . . .	51
6.1.3	First quantum circuit example . . . . .	52
<b>7</b>	<b>QKD in softwarised networks</b>	<b>55</b>
7.1	Introduction . . . . .	55
7.2	Use cases . . . . .	56
7.2.1	Two instances key exchange . . . . .	56
7.2.2	Multi instances key exchange . . . . .	58
<b>8</b>	<b>QKD simulator</b>	<b>61</b>
8.1	Simulator Design and Architecture . . . . .	61
8.2	Simulated protocols . . . . .	64
8.2.1	BB84 . . . . .	64
8.2.2	E91 . . . . .	65
8.3	Interfaces . . . . .	66
8.4	Criticalities . . . . .	70
8.5	Workflow . . . . .	71
<b>9</b>	<b>QKD Module (QKDM)</b>	<b>76</b>
9.1	QKD Module Design and Architecture . . . . .	76
9.2	Interfaces . . . . .	78
9.3	Criticalities . . . . .	80
9.4	Workflow . . . . .	81

<b>10 QKD Key Server</b>	84
10.1 Key Server Design and Architecture . . . . .	84
10.2 Interfaces . . . . .	88
10.3 Criticalities . . . . .	90
10.4 Workflow . . . . .	91
<b>11 Test and validation</b>	96
11.1 QKD Simulator . . . . .	96
11.2 QKD Key Server and Module . . . . .	99
<b>12 Conclusions</b>	103
<b>Bibliography</b>	104
<b>A User Manual</b>	109
A.1 QKD Simulator . . . . .	109
A.2 QKD Key Server and Module . . . . .	115
<b>B Developer’s Manual</b>	121
B.1 QKD Simulator . . . . .	121
B.2 QKD Key Server and Module . . . . .	126

# Chapter 1

## Introduction

Quantum computing is one of the most active field of research today. Interest in this technology is due to the promise it carries with it: a computational power beyond any other classical computer. This is because quantum computers can elaborate information differently compared to classical computers. The importance of this technology lies on the fact that, for particular applications, they are capable of solving given tasks, that for classical computers are considered to be extremely complex, in a reasonable time.

With these premises different companies started to develop quantum computing based technologies in order to achieve the so called *quantum supremacy*. In particular, companies like IBM and Google invested in quantum research and developed their first prototypes of quantum computers. Recently IBM announced a roadmap in which they foresee to build a 1121-qubit quantum computer by 2023<sup>1</sup>.

Nevertheless, the rising of quantum computing yield to new concerns that were not considered before, in particular for those applications which lean on long computational tasks to ensure a desired result. This is the case of classical cryptography in which some application, like key exchange, are considered safe because it is actually impossible for a classical computer to solve operations on which it is based in a reasonable time. If quantum supremacy will become true these assumptions cannot be longer considered true and the cryptography on which all systems are based on today cannot be longer considered safe.

Contextually to quantum computers development a new field of research has unfolded aimed to solve the same problems that arise from quantum computing. In particular *Quantum Key Distribution* (QKD) is a field of research that tries to efficiently and securely solve key exchange challenges ensuring security against both classical and quantum computers. Protocols and examples of quantum key distribution have been successfully realized between distant nodes. However, the low diffusion of QKD devices requires multiple applications to share the same technology in order to ensure a secure key exchange even before spread of quantum technology will be achieved.

The proposed work is aimed to give an overview on quantum key distribution, focusing on the principles that allow to guarantee a secure key exchange against all adversary. The main protocols and standards will be discussed, outlining the main criticalities of practical implementations. Moreover, a QKD model for softwarised network will be proposed, based on a simulated QKD environment.

---

<sup>1</sup><https://www.sciencemag.org/news/2020/09/ibm-promises-1000-qubit-quantum-computer-milestone-2023>

## Chapter 2

# Quantum Computing

### 2.1 Quantum Information and Quantum computing

Quantum information refers to the information about the state of a quantum system. If classical information is processed through digital computers, quantum information can be processed using quantum computing. With quantum computing it is possible to manipulate quantum information to solve some category of problems in a relative small amount of time compared to time required by classical computers. This is done by exploiting quantum mechanical principles including superposition, interference, entanglement, no-cloning theorem and decoherence. All these principles and how they can be used in quantum computing will be explained in this chapter but, before introduce them, it may be worth to focus the attention on the basic unit of quantum information to which these principles apply: the qubit.

#### 2.1.1 Qubit

A qubit is the basic unit of information in quantum computing, just like bit for classical computing. The main difference is that quantum computing manipulates elemental particles that obey the quantum mechanics laws and makes measurements on some of their properties to represent value 0 or 1. As example, 0 and 1 values can be encoded on the spin of an electron as shown in figure 2.1. Before a measurement takes place and a value (0 or 1) is assigned, the qubit is said to be in a superposition of states: in classical information a bit can have the value '0' or '1'; a qubit instead can represent both of this two states at the same time. This is possible because the state of a qubit, the quantum state, is something more complex than the classical binary values.

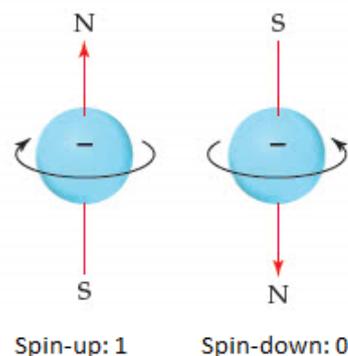


Figure 2.1. Electrons spin associated to bit values 0 and 1 (source: [oneclass](#)).

The quantum state provides a probability distribution for the outcomes of each possible measurement on a system. In such scenario, state 0 can be seen as the state in which the measurement

of a qubit definitely gives the outcome 0, whilst state 1 is the state in which measurement definitely gives the outcome 1. From a mathematical perspective a qubit state can be described as a unit vector in two-dimensional Hilbert space [1]. Hence, quantum states 0 and 1 can be described as follows:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (2.1)$$

Vector  $|0\rangle$  and  $|1\rangle$  are two orthonormal basis state. A superposition of states has the meaning of a linear combination of such states. Hence, the state of any qubit can be described as follows [2]:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \quad (2.2)$$

where  $\alpha$  and  $\beta$  are probability amplitudes that can also be complex numbers.

Since  $\alpha$  and  $\beta$  represent probabilities, it is possible to state that a qubit is described by the probability ( $\alpha$ ) to have the value “0” and the probability ( $\beta$ ) to have the value “1”, hence it is actually both of them at the same time. Only after measurement the qubit will collapse to one of the two values.

Please note that  $|0\rangle$  and  $|1\rangle$  are not the only orthonormal basis that can be used to represent a qubit state. Any pair of orthonormal vectors can be linear combined to represent the qubit. This means that only when measured with basis  $|0\rangle$  and  $|1\rangle$  the state of qubit will collapse to the canonical 0 or 1 value.

Since any orthonormal vectors pair can be used to represent the state of a qubit there are potentially infinite basis. Besides  $|0\rangle$  and  $|1\rangle$  other common basis are:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad (2.3)$$

and

$$|i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) \quad |-i\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle) \quad (2.4)$$

Besides representing quantum states with vectors in Hilbert space, there is also the possibility to express probabilities  $\alpha$  and  $\beta$  as real numbers, adding a term that state the relative phase between them. This leads to the following definition of qubit state [3]:

$$|\psi\rangle = \cos\frac{\theta}{2} |0\rangle + e^{i\phi} \sin\frac{\theta}{2} |1\rangle \quad (2.5)$$

Where  $\phi \in [0, 2\pi]$  describes the relative phase and  $\theta \in [0, \pi]$  determines the probability to measure  $|0\rangle$  or  $|1\rangle$ .

This notation is particular convenient since by interpreting  $\phi$  and  $\theta$  as spherical co-ordinates it is possible to have a visual representation of the quantum state. This is the so called Bloch-sphere notation.

When working with this notation, it is important to notice that on the bloch-sphere angles are twice as big as in Hilbert space as can be seen from the picture above: basis  $|0\rangle$  and  $|1\rangle$  are orthogonal, but on the bloch-sphere their angle is  $180^\circ$ . Hence if  $\theta$  is the angle on the bloch-sphere the actual angle is  $\frac{\theta}{2}$  in Hilbert space.

### 2.1.2 Bra-ket notation

As already seen in previous paragraph, qubits can be represented by vectors. To simplify the way they are handled, a compact notation has been introduced: the bra-ket notation. This notation has been introduced in 1939 by Paul Dirac [4], hence it is also known as Dirac notation.

From a mathematical point of view a ket represents a vector in a complex vector space (Hilbert space). Its notation consists of a vertical bar  $|$  and a right angle bracket  $\rangle$ . e.g:  $|0\rangle$ . It is used

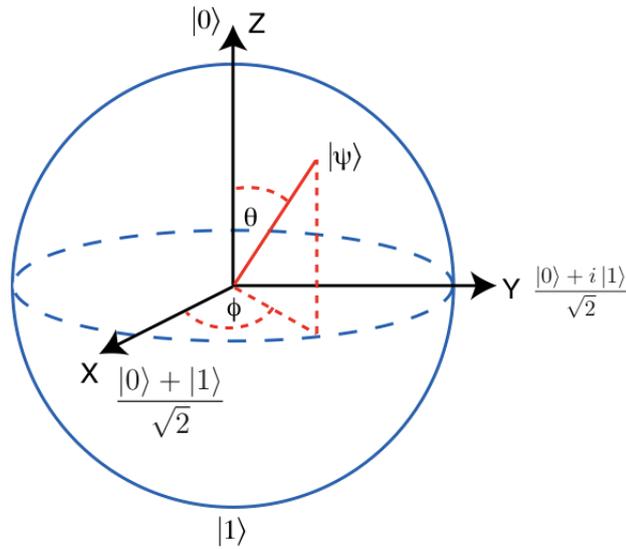


Figure 2.2. Graphical quantum states representation on Bloch-sphere (source: [qiskit](#)).

to represent quantum states. A bra is used to represent a linear map that maps each vector to a complex number. Mathematically, it is the complex conjugate transposed of a ket. Its notation consists of a left angle bracket  $\langle$  and a vertical bar  $|$ . e.g:  $\langle f|$ . Basically ket represents column vectors and bra row vectors.

From the definition of bra and ket it is possible to define the bra-ket operation, that basically is the inner product of two vectors:

$$\langle 0|1\rangle = [ 1 \quad 0 ] \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1 \cdot 0 + 0 \cdot 1 = 0 \quad (2.6)$$

Please note that above result is 0 because  $|0\rangle$  and  $|1\rangle$  are orthogonal basis.

Ket-bra notation refers instead to the tensor product of vectors. This can be useful when it is needed to describe the state of more qubits together:

$$|ba\rangle = |b\rangle \otimes |a\rangle = \begin{bmatrix} b_0 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \\ b_1 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} b_0 a_0 \\ b_0 a_1 \\ b_1 a_0 \\ b_1 a_1 \end{bmatrix} \quad (2.7)$$

### 2.1.3 Superposition

Superposition has already been described as a linear combination of two quantum states (eq.(2.2)). However, this is just a mathematical definition. From the physics perspective superposition can be seen as two quantum states added together, generating a new quantum state. Comparing it to classical physics, it is like two waves added together to create a new third wave.

Considering that each quantum state has its own amplitude, when two quantum states are superposed interfere each other in a constructive or destructive way. Thinking again at waves, a constructive interference is when two waves with the same phase are added together: in this case the energy of the resulting wave will be doubled. Instead, If two waves completely out of phase are superposed, the energy of the resulting wave will be zero. In this case there is a destructive interference.

Quantum interference is the basis principle of quantum computing. By letting two particles interfere it is possible to bias the measurement of a qubit toward a desired state. In other words

when a quantum system has to solve a problem it has to handle a pattern of interference in which the paths leading to wrong solutions interfere destructively and cancel out while the paths leading to the right solution interfere constructively maximizing energy of the state corresponding to the right solution.

Quantum interference must not be misunderstood with decoherence. While interference is an induced phenomenon for producing a desired effect on the qubit state, decoherence regards the collapse of the quantum state due to interference with external environment. In other words decoherence is an undesired disturb on the quantum state that lead to losing superposition properties.

### 2.1.4 Entanglement

Quantum entanglement is a particular phenomenon that involve two or more particles. Theoretically there is no limit to the number of particle that can be entangled together. In practice, since it is a complex procedure, it is usually performed with a pair of particles.

When quantum entanglement takes place, a pair of particles physically interact and bound each other. After this process the quantum state of a particle is highly correlated to the other one in a way such that any action performed on one affect the other one. This bond is still valid even if the two particles are placed at great distances. This implies that a measurement on a particle will give information on the state of the entangled particle, even if the two are isolated and far from each other. By exploiting this principle entangled particles can be sent to different people and each one can know the state of the other particles by looking at the outcomes of his measurements.

Mathematically two entangled state can be expressed as follows:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \quad (2.8)$$

The meaning of this formula is that measurements on these state can lead to result  $|00\rangle$  or  $|11\rangle$  with a probability of 50%. Instead there is no chance qubits can be found in  $|01\rangle$  or  $|10\rangle$  states. Thus, if just one qubit is measured and found in  $|0\rangle$  state the other qubit will be in  $|0\rangle$  state with a probability of 100%. The same result can be found if measurement outcome of the first qubit is  $|1\rangle$ : when the second qubit will be measured it will certainly be  $|1\rangle$ .

For two qubits, another entangled state is the following:

$$\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \quad (2.9)$$

In which measurement outcome will be opposite for the two qubits.

Concept of entanglement is something related to quantum mechanics only. There is no similar concept in classical physics. It seems to violate the theory of relativity from which can be derived that the maximum speed limit of information transmission is the speed of light (actually since there is no information transferred during measurement this has been proved to be false [5]). To explain this behaviour in 1935 Einstein, Podolsky and Rosen proposed the hidden variable theory. This theory introduces unobservable hypothetical entities to provide a deterministic explanation of quantum mechanics phenomena. With such variables it would be possible to predict quantum measurement results on a given particle. However, in 1964, Bell proposed a theorem to prove that quantum physics is incompatible with hidden variable theory [6]. Specifically Bell demonstrated that correlation resulting from entanglement cannot be explained with hidden variable theory, otherwise the Bell's inequality, that regards measurements made on entangled particles pairs, would be violated. Bell's inequality has been generalized by CHSH inequality, which remove any assumption of perfect correlation of the entangled particles. The CHSH inequality can be used as indicator of quantum entanglement to check if two particles have entanglement properties.

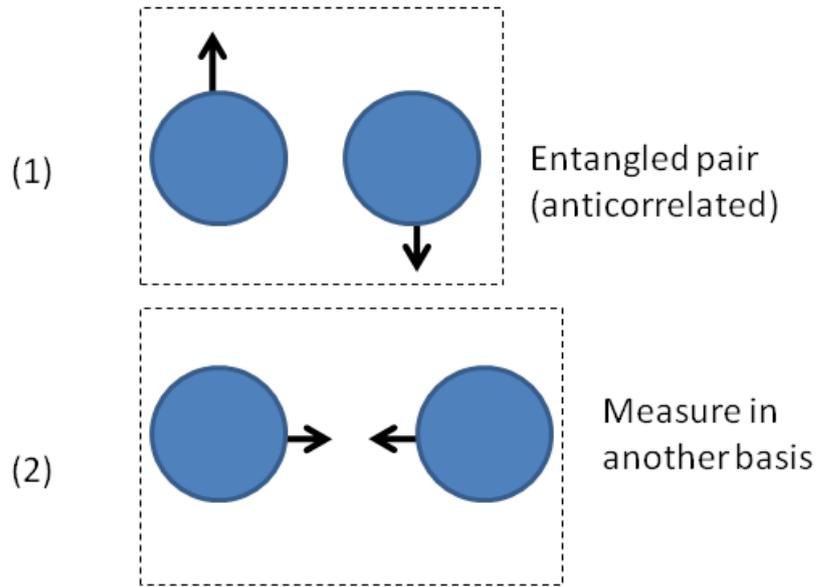


Figure 2.3. Instantaneous effect of measurement on entangled qubits.

### 2.1.5 Measurement

Measurement in quantum mechanics regards the manipulation of a quantum system in order to get a numeric value from one of its properties. Once measured the original qubit status is lost and the qubit will collapse to the measured state.

Practically, measurement is based on the *Born's rule* which allows to calculate the probability that a given quantum state,  $|\psi\rangle$  collapses to a generic state  $|x\rangle$ . This probability is given by [7]:

$$p(x) = |\langle x|\psi\rangle|^2 \quad (2.10)$$

Measurement can be performed on any orthonormal basis. The measurement result will be one of the two state vectors forming the basis. As example, considering basis  $|0\rangle$  and  $|1\rangle$ , measure a qubit in this basis will let it collapse to  $|0\rangle$  or  $|1\rangle$ . No other result except from these two can outcome from the measurement. Basically measurement maps the quantum state to one of the basis vector. This kind of measurement is called *projection measurement* [8].

An important property of qubit is that, if it is in a known state and measurement is done in an orthogonal basis, the outcomes are completely random. For a qubit in superposition state, choosing basis  $|0\rangle$  and  $|1\rangle$ , the probability for a qubit to be measured in  $|0\rangle$  state is the same as the probability to be measured in  $|1\rangle$  state: 50%. This can be proven considering a generic state vector representing a qubit state:

$$|\psi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{bmatrix} \quad (2.11)$$

As already stated, any qubit state can be expressed as linear combination of orthonormal basis. Hence the  $|\psi\rangle$  qubit can be written as follows:

$$|\psi\rangle = \frac{1}{\sqrt{2}} |0\rangle + \frac{i}{\sqrt{2}} |1\rangle \quad (2.12)$$

Applying Born's rule it is possible to calculate the probability of measuring qubit  $|\psi\rangle$  in state  $|0\rangle$ :

$$p(0) = |\langle 0|\psi\rangle|^2 \quad (2.13)$$

where:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{i}{\sqrt{2}}|1\rangle \quad (2.14)$$

$$\langle 0|\psi\rangle = \frac{1}{\sqrt{2}}\langle 0|0\rangle - \frac{i}{\sqrt{2}}\langle 0|1\rangle \quad (2.15)$$

$$= \frac{1}{\sqrt{2}} \cdot 1 - \frac{i}{\sqrt{2}} \cdot 0 \quad (2.16)$$

$$= \frac{1}{\sqrt{2}} \quad (2.17)$$

hence:

$$p(0) = |\langle 0|\psi\rangle|^2 = \frac{1}{2} \quad (2.18)$$

The same result can be found calculating the probability of the outcome  $|1\rangle$ :

$$p(1) = |\langle 1|\psi\rangle|^2 \quad (2.19)$$

where:

$$\langle 1|\psi\rangle = \frac{1}{\sqrt{2}}\langle 1|0\rangle - \frac{i}{\sqrt{2}}\langle 1|1\rangle \quad (2.20)$$

$$= \frac{1}{\sqrt{2}} \cdot 0 - \frac{i}{\sqrt{2}} \cdot 1 \quad (2.21)$$

$$= -\frac{i}{\sqrt{2}} \quad (2.22)$$

hence:

$$p(1) = |\langle 1|\psi\rangle|^2 = \frac{1}{2} \quad (2.23)$$

If, for a qubit measured in an orthogonal basis, a measure will outcome a completely random result, after measurement the state of the qubit is known and will remain that for any further measurement that use the same basis. In other words, when a qubit in superposition state is measured there is the 50% of probability to have status 0 and 50% to get status 1. After measurement, if qubit has been found to be in status 0(1), all the subsequent measurements with the same basis vectors will outcome 0(1) with a probability of 100%. This behaviour is the same for any basis the measurement is performed on: the qubit “choose” one of the two state vectors composing the basis when measured and keep this value for any further measurements with the same basis.

A different consideration must be done if the same qubit is measured more than once with different basis. As said, measurement cancels out previous qubit state (unless the same basis are used in subsequent measurements). That said, if a qubit is measured in a basis and takes a given value with a probability of 50% a new measurement in an orthogonal basis will outcome one of the two status composing the basis with a probability of 50% again. Furthermore, qubit memory is limited to the last performed measurements: if a qubit is found to be in  $|0\rangle$  status after a measurement, any new measurement in another basis will cancel that information and, when the qubit is measured again in  $|0\rangle$  and  $|1\rangle$  basis, the measurement outcome will again be uncertain, with the same probability (50%) of be either  $|0\rangle$  or  $|1\rangle$ .

## 2.2 Quantum computing

Quantum computing manipulate qubits, performing operations on them. Measurement is usually one of the latest operation performed on a qubit, since it destroys its state. Qubit are hence manipulated without being observed by means of gates.

Gates are operators that transform a qubit in another one by applying a specific operation. Mathematically they can be seen as matrices that applied to the qubit vector state modify its

value. Due to quantum mechanics properties, contrary to operations on classical computation, gates operations are reversible.

Gates are usually divided into single-qubit gates and multi-qubit gates, depending whether they operate on a single qubit or on multiple qubits at the same time.

Here is reported a table with the most commonly used gates in quantum computation: Only

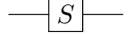
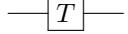
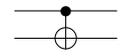
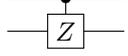
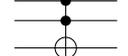
Name	Symbol
Pauli-X (X)	
Pauli-Y (Y)	
Pauli-Z (Z)	
Hadamard (H)	
Phase (S)	
$\frac{\pi}{8}$ (T)	
Controlled Not (CNOT)	
Controlled Z (CZ)	
SWAP	
Toffoli (CCNOT)	
Measurement	

Table 2.1. Gates list.

the most important of them will be discussed here. The others are left to further readings<sup>1</sup>.

### 2.2.1 Single-qubit gates

Among the single-qubit gates it may worth to mention those belonging to a particular family of gates known as Pauli gates. Pauli gates are probably the most commonly used quantum gates, mainly because they allow performing simple transformation on qubits. X, Y and Z gates belongs to this family.

X gate is one of the fundamental quantum gates. It is used to flip the state of a qubit to 1 if it is 0 or 0 if it is 1. Basically it flips the amplitude of the qubit and it can therefore be considered as a NOT port in classical computation. It is represented by the following matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.24}$$

Its behaviour can be observed by applying it to status  $|0\rangle$  and check that result correspond to status  $|1\rangle$ :

$$X |0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle \tag{2.25}$$

Similarly it is possible to apply X gate to  $|1\rangle$  status:

$$X |1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle \tag{2.26}$$

---

<sup>1</sup><https://qiskit.org/textbook/ch-states/single-qubit-gates.html>

If X gate flips the amplitude of a qubit, Z gate flips its phase. It is represented by the following matrix:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad (2.27)$$

The behaviour of this gate on states  $|0\rangle$  and  $|1\rangle$  can be easily seen with this trivial calculation:

$$\begin{aligned} Z|0\rangle &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle \\ Z|1\rangle &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} = -|1\rangle \end{aligned} \quad (2.28)$$

Y gate is instead represented by this matrix:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad (2.29)$$

Applying this gate to states  $|0\rangle$  and  $|1\rangle$  leads to the following results:

$$\begin{aligned} Y|0\rangle &= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ i \end{bmatrix} = i|1\rangle \\ Y|1\rangle &= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -i \\ 0 \end{bmatrix} = -i|0\rangle \end{aligned} \quad (2.30)$$

Basically, this gate has the effect of flip both amplitude and phase of the qubit.

Besides Pauli gates, another important gate that is worth to mention is the Hadamard (H) gate. This gate is defined as follows:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (2.31)$$

To check its behaviour it is possible applying it to state  $|0\rangle$  and  $|1\rangle$ :

$$\begin{aligned} H|0\rangle &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ H|1\rangle &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \end{aligned} \quad (2.32)$$

The importance of this gate can be noticed if the resulting states are expressed by means of a linear combination of  $|0\rangle$  and  $|1\rangle$  states and the probability of getting state  $|0\rangle$  or  $|1\rangle$  upon measurement is calculated (the same calculations shown in equations (2.13) and (2.19) can be applied). In this case it can be noticed that probability of getting state  $|0\rangle$  is the same as the probability of getting state  $|1\rangle$ : 50%. This means that applying the Hadamard gate allows to generate a superposition of  $|0\rangle$  and  $|1\rangle$  states.

### 2.2.2 Multi-qubit gates

Multi-qubit gates act on groups of qubit at the same time. They are important because to gain a computational advantage from quantum computing qubits must be processed together. If a qubit is represented by two complex amplitudes, two qubits are represented by four complex amplitudes as follows:

$$|\alpha\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle = \begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix} \quad (2.33)$$

One of the most known multi-qubit gate is the Controlled NOT (CNOT). This gate can be used on 2 or more qubits together. Considering just two qubits It can be schematized by the following matrix:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad (2.34)$$

Applying this gate to the classical basis has the following effect:

$$CNOT |\alpha\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix} = \begin{bmatrix} \alpha_{00} \\ \alpha_{11} \\ \alpha_{10} \\ \alpha_{01} \end{bmatrix} \quad (2.35)$$

An easier way to understand its behaviour is to look at the related truth table:

Input (t,c)	Output(t,c)
00	00
01	11
10	10
11	01

Table 2.2. CNOT truth table.

To describe its effect, consider that the first qubit is the target whilst the second one is the control qubit. When the two qubits are in the standard basis  $|0\rangle$  and  $|1\rangle$  the gate applies the NOT operation on the target if the control qubit is  $|1\rangle$  otherwise the target is left unchanged as can be seen from the above calculations.

The real importance of this gate is outlined when the qubits it operates onto are in superposition state. As example consider CNOT gate applied to qubits  $|0+\rangle$ . Calculating the tensor product of these qubits the resulting state vector is the following:

$$|0+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle) \quad (2.36)$$

Applying CNOT gate will result in:

$$CNOT |0+\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \quad (2.37)$$

As result, CNOT gate has generated entanglement between qubits in input. This gate is particular important because, up today, it is the only gate that allows to generate entanglement between qubits.

### 2.2.3 Quantum circuit

All qubit gates can be combined in a so called quantum circuit to get the desired result. The advantage of quantum circuits is that, instead of working with matrices, it is possible to have a graphical representation of the gates and how they interact each other. This simplifies qubits manipulation from the perspective of users.

In a quantum circuit each line corresponds to a qubit. Upon these lines gates are applied. As example consider the following X gate applied to a generic qubit:

$$|0\rangle \text{ --- } \boxed{X} \text{ --- } |1\rangle$$

Multi-qubit gates involve the lines relative to the qubits they affect. As example consider the following CNOT gate where the first line represents the control qubit and the second one the target:

$$\begin{array}{ccc} |1\rangle & \text{---} \bullet & |1\rangle \\ |0\rangle & \text{---} \oplus & |1\rangle \end{array}$$

Here is presented an example of a simple quantum circuit:

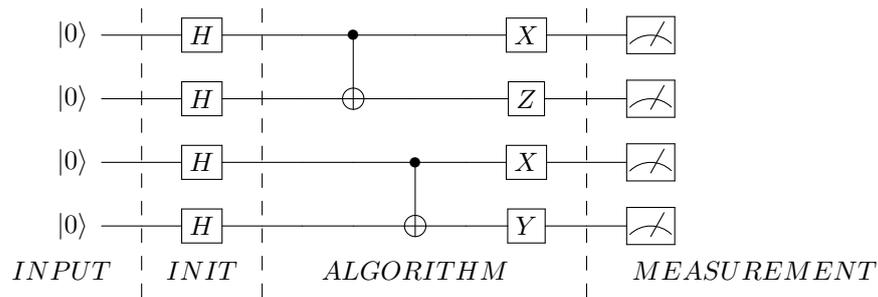


Figure 2.4. Simple quantum circuit.

As can be seen from figure 2.4, a quantum circuit can be seen as an ordered execution of the following phases:

- *Input*: the input of a quantum circuit is usually qubits in a classical state ( $|0\rangle$  by convention).
- *Initialization*: the quantum algorithm is applied on the inputs that the most of the time need to be manipulated in order to be in superposition state. Hence input needs to be prepared before it can be processed by the quantum algorithm.
- *Algorithm*: quantum algorithms consist of applying gates in order to create the right interference on the inputs. After the algorithm is applied, the final state will have most of its weight on solution to the treated problem.
- *Measurement*: to check what this solution is the final stage of a quantum circuit is measurement. Measurement will result again in a classical state that can be traced to classical bit 0 or 1.

Inputs and outputs of a quantum circuit are nothing more than classical 0 and 1 bits. The quantum phenomena come into play after inputs preparation phase, when quantum algorithm is applied.

## 2.3 From classical to quantum cryptography

The need for a secure communication even in the presence of an adversary who may intercept the information exchanged is an ancient problem that has been tried to be solved for centuries. Cryptography, by means of encryption, tries to solve this problem by defining techniques and procedures that continuously evolve with the progress of technology. This section gives a quick overview of the current status of classical cryptography, outlining the main reasons behind the development of quantum cryptography.

### 2.3.1 Theoretical secure communication

Study of cryptography over decades led to an important result: a perfectly secure communication is actually possible (at least in theory).

A perfectly secure communication is a communication that cannot be decrypted from an adversary, even if he has unlimited computational power. This theory has been studied and developed by Claude Shannon who in 1949 proved that one time pad cryptosystems are theoretically secure encryption systems [9]. What Shannon outlined is that a perfect security can be achieved by employing one time pad encryption algorithm.

One time pad algorithm is a symmetric encryption scheme that uses a key of the same length of the cleartext. The principle of operations is simple: each bit of the key is XORed with each bit of the cleartext. The so encrypted text can then be sent to the receiver who will decrypt it by XORing the ciphertext with the same key. This is enough to guarantee a theoretical security if and only if the key is truly random and it is never reused in whole or in part. In other words a completely different keys must be used for each message that has to be exchanged. In such a scenario the number of keys to be used is equals to the number of messages that has to be exchanged that may potentially be infinite. This also means that preshared keys is not a feasible solution. The only way to ensure different keys for different messages is to exchange the keys when needed. Hence, the problem of ensuring a secure communication translates into the need of securely exchange keys.

### 2.3.2 Classical cryptography threats

Modern encryption systems can basically be divided into *symmetric* and *asymmetric* systems.

In symmetric encryption the involved parties communicate through the same key, that they use to manipulate the message in such a way that it is impossible for whoever read the message to understand the information that it carries. Since the same key is used for encryption and decryption, each distinct pair of parties that can be involved in a communication must share different keys. This leads to an exponential growth of the number of keys to be managed.

Asymmetric encryption (also known as Public Key encryption) uses a pair of keys bounded together by a mathematical relationship. One of this key is called public key and can be freely distributed to anyone. The other one is called private key and, as the name suggests, it must remain a secret. The advantage of asymmetric encryption is that anyone can generate a secret message for a given subject by using a public information (his public key). The effective strength of the asymmetric encryption lies on the mathematical relationship chose to bind public and private key. One of the most important features these keys have to guarantee is that the calculation of the private key must be infeasible starting from the public key. To ensure this property asymmetric encryption algorithms use operations such as large numbers factorization, exponentiation or logarithm. All these operations, if the number of bits of the key is large enough, require a considerable amount of time (in the order of years or more) to revert a single key with computational power of current computers. Hence security is entrusted to the difficulties of current computers to solve certain mathematical problems.

Nowadays asymmetric encryption is one of the most adopted techniques to exchange keys. Most used protocols allow to derive a key from already owned information removing the needs

of a real key exchange. An example is Diffie-Hellman algorithm that allows to negotiate a private and public key pair by exchanging just public information. In practice this algorithm uses mathematical operations such as modulus to derive the key pair. The security leans again on the impossibility for current computers to solve the reverse operations.

Advent of quantum computing yield to important concerns for classical asymmetric encryption. Quantum computers are capable of solving some categories of problems in polynomial time. This translates into the possibility for a quantum computer to retrieve private key the from public one if certain kind of asymmetric encryption algorithms are employed. An example can be found in Shor's algorithm [10]. Shor's algorithm is a quantum algorithm capable of solving integer factorization problems in polynomial time. More specifically, given an integer it finds its prime factors. This means that by using a quantum computer running Shor's algorithm it is possible to break asymmetric encryption schemes that lean on large numbers factorization (such as the widely used RSA algorithm) in a reasonable time.

Technology to build quantum computers is still immature but it is rising fast. In the latest years, interest in quantum computing is increased and quantum computing has been inserted in the list of top emerging technologies of Gartner 2018 report [11]. Current quantum systems are capable of managing just few qubits (the current machines are capable of manage up to 53 qubits <sup>2</sup>) mainly for the difficulty of isolate this kind of systems (any interference can destroy the quantum state). However, as already said in introduction, producers like IBM foresee quantum systems capable of managing more than 1000 qubits by 2023, proving the fast growing of this technology. With current systems, Shor's algorithm cannot be considered a threat today. A recent study tried to factor a 2048 RSA integer with Shor's algorithm in a simulated environment [12]. The results proved that time required by this algorithm is just 8 hours, but 20 million of noisy qubits were employed. This study shows how today it is already possible to estimate the measure of threat represented by quantum systems to classical cryptography, even if it is not currently employable in real contexts.

Even if quantum computers are not a spread reality today, concerns for classical cryptography started to increase in the latest years. This led to the study of possible countermeasure and methods to effectively achieve a secure communication.

### 2.3.3 Post quantum cryptography

Post quantum cryptography refers to a kind of cryptography that it considered to be secure against quantum computers attacks. There are different approaches to realize post quantum cryptography. The most important can be categorized in: lattice-based, hash-based, code-based and multivariate polynomial cryptography.

Lattice-based cryptography is a technique based on lattices. A lattice is the set of all integer linear combinations of basis vectors. This category of algorithms is based on the hardness of the shortest vector problem that have been shown to be hard to solve efficiently even with approximation factors that are polynomial and even with quantum computers [13]. One of the most famous lattice-based encryption scheme is NTRU. NTRU is based on the difficulty of factoring certain polynomials in a truncated polynomial ring into a quotient of two polynomials having very small coefficients. Since it still relies on mathematical problems, its parameters must be carefully chosen in order to avoid known attacks. Implementation of NTRU has been proposed to replace RSA and ECC encryption scheme.

Hash-based cryptography construct cryptographic primitives based on hash functions. Security is hence strictly bind to the chosen hash function, in particular to hash functions properties such as collision resistance and pre-image resistance [14]. Current development is limited to to digital signatures schemes. A famous hash-based encryption scheme is SPHINCS, that in 2017 has been improved by NIST in the currently proposed SPHINCS+ <sup>3</sup>.

---

<sup>2</sup><https://www.cnet.com/news/ibm-new-53-qubit-quantum-computer-is-its-biggest-yet/>

<sup>3</sup><https://sphincs.org/>

Code-based cryptography is based on error-correcting codes and it is suitable for public key encryption. Quite all implemented algorithms in this class use large keys to ensure security. An example of code-based cryptography is McEliece public key encryption scheme [15].

Multivariate polynomial cryptography relies on the difficulty of solving the multivariate polynomial algorithm over finite fields. This class of algorithms has been found to be particularly suitable for building signature schemes since it provides extremely short signatures [16].

By replacing currently available encryption schemes with the proposed post quantum algorithms it is possible to secure communications from quantum computers advent. However, they still entrust security on complexity of mathematical functions. Besides these algorithms are relatively new and have been studied much less than classical cryptographic algorithms, hence it cannot be excluded that somewhere in the future an algorithm capable to break them will be found.

### 2.3.4 Quantum Cryptography

In order to solve keys exchange problem it would be important to find a way for exchanging keys that does not lie on the complexity of reversing some mathematical relationship. Quantum cryptography tries to solve this problem exploiting the already seen properties of quantum systems. Quantum cryptography can be summarized in different categories:

- Fully quantum cryptography: it makes exclusively use of quantum system to ensure security. An example of this is quantum one time pad algorithm (QOTP).
- Hybrid solutions: uses both quantum and classical communication to achieve security. QKD and quantum money are examples.
- Beyond QKD: basically it refers to the attempts of using quantum cryptography to solve problems that cannot be solved employing classical cryptography.

#### Quantum one time pad

QOTP is a symmetric encryption scheme that work similarly to the classical one time pad algorithm. The only difference is that the XOR operation between cleartext and key is here replaced by a quantum operation consisting of applying X and Z Pauli gates. Basically if in classical one time pad XOR operation is applied both for encrypt and decrypt operation, here X and Z gate must be applied in an order when encrypting and reverse order when decrypting. An example of a circuit that allows to transform a generic qubit  $|\psi\rangle$  in a ciphered qubit  $|c\rangle$  is shown in figure 6.1. In order to revert encrypt operation and recover the original information X and Z gates must be

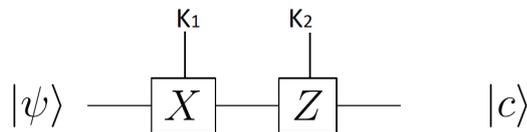


Figure 2.5. QOTP encryptor.

applied in reverse order as shown by circuit in figure 2.6.

In such scenario the key is used as control for X and Z operation. In particular, here two control bits,  $k_1$  and  $k_2$ , are required for each qubit that must be encoded. Each bit will control activation of a gate, hence the first bit will control activation of the first gate, enabling it if it is one or disabling it otherwise, as well as the second bit will do for the second gate. In this way only who has the key can perform the correct sequence of operations that must be performed on each qubit to recover the original information.

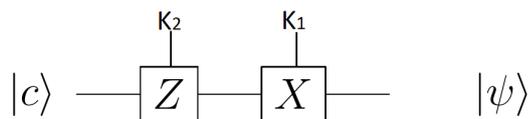


Figure 2.6. QOTP decryptor.

In practice quantum one time pad does not lead to any advantage compared to classical one time pad. On the contrary, it needs the key to be double length of the cleartext in order to control two gates for each qubit, while classical one requires the key to be just long as the cleartext. This is why even if it is one of the few examples of fully quantum cryptography it is not used for practical applications.

### Hybrid solutions

Another field of study in quantum cryptography is the one that make use of quantum systems properties to ensure security, but it is not fully quantum since it needs to post process information exchanged through classical communication channels in order to realize its protocols. In this category it is possible to find quantum key distribution and quantum money.

Quantum Key Distribution is the most promising application of quantum cryptography. It tries to solve key distribution problem in such a way that security does not lean on some mathematical relationship, but on some peculiar properties of quantum systems. In particular, it exploits the fact that, if a qubit is measured by any adversary, the qubit states is destroyed and the two parties can be aware of the eavesdropper presence. This outlines the main important difference between classical and quantum encryption.

In classical encryption an adversary may sniff the communication and store exchanged information. Then, some time forward in future he can use a computer with unlimited computational power or exploit an algorithm that was not known at the moment of the exchange to break the security. In such a scenario, parties involved in the communication will never know that someone has been able to steal their information. With quantum key distribution, if an adversary tries to sniff qubits exchange, the exchange itself fails. This means that if a key is correctly exchanged the two parties know that a theoretical truly secure communication can take place. Protocols and technology that makes this possible will be studied in deep in the next chapter.

Another interesting application of quantum cryptography is quantum money. The aim of this technique is to exploit quantum systems properties to create a currency that cannot be forged. The principle of operations is to assign a unique serial number to each bank note together with a series of two-state quantum bits. For each serial number bank stores a record of all the polarizations of the corresponding bank note in the correct base. Hence the bank is the only entity that can measure the polarization of the qubits in the bank note without destroy their state. This means that bank can always verify the bank note authenticity. An eventual forger instead does not know the correct polarization the qubits must be at and the only thing he can do is guessing. If the total number of qubits in the bank note is large enough the probability he is able to guess all polarizations becomes exponentially small [17]. Although quantum money is a promising application, there are no practical implementations because of limitations of current quantum devices that can store quantum states only for few time.

### Beyond QKD

As said quantum key distribution is the most developed topic in quantum cryptography. However, many other different techniques have been developed that try to solve known problem in classical cryptography. In particular some of these techniques regards: *quantum digital signatures*, *blind quantum computing* and *bit commitment*.

The need of quantum digital signature schemes is born from the same concerns that led to quantum key distribution: in classical digital signature schemes security is based on the assumption that some mathematical problems are considered hard to solve. This leads to study of schemes based on quantum properties. The first proposed scheme is Gottesman-Chuang QDS [18]. Although this protocol is not realizable because of the need of long time quantum memory, it raised a lot of interest on the subject and lead to other developments that have been successfully implemented [19].

Blind quantum computing tries to solve the privacy problem when a computing task is delegated to an untrusted device. The assigned device must be able to compute quantum tasks without accessing data it has to manipulate. Even for this problem, experimental protocols have been proposed [20].

Bit commitment take place between two mistrustful parties. Basically one of the two parties commits the value of a bit and sends it to the other one. The receiver cannot check the value of the bit until the sender reveals it, at the same way the sender cannot change bit value after commitment. Bit commitment has been proved to be impossible to achieve [21]. However some experimental bit commitment scheme was implemented under particular physical assumptions, for example that quantum memory of attacker is noisy [22].

## Chapter 3

# Quantum Key Distribution

### 3.1 QKD overview

Quantum Key Distribution is one of the most promising applications of quantum cryptography. As already said, it aims to solve the problem of a trustable key distribution without lean on complex mathematical relationship that are known to not be easily solved with current technology, but may be solved in future with technological progress.

Relying on quantum mechanics, QKD bases its security on the fact that a measurement on a qubit destroys its quantum state. This ensures that, if an eavesdropper tries to measure exchanged qubits to understand their value, he unavoidable changes qubits status if he measures with basis that are not foreseen by the two parties of the communication. Hence, applying specific protocols, sender and receiver can notice that qubits measurements led to different results from the expectations and be aware of the presence of an eavesdropper.

QKD claims to be the final solution for a theoretically secure communication. In this chapter its main properties and some practical protocols will be outlined to understand the reason behind this assumption.

To briefly summarize quantum key distribution is composed of different building blocks that will be discussed here:

- No-cloning principle: it is the fundamental criterion on which QKD is based.
- Random number generators: used to generate truly random keys.
- Photon sources: used to create the quantum particles that will carry information.
- Encoder: used to encode information on created quantum particles.
- Quantum channel: the mean that will transmit quantum particles.
- Detectors: used to intercept and measure the particles on quantum channel.
- Classical channel: used to post-process information exchanged.
- Quantum protocols: orchestrators that use all previous building blocks to perform a secure QKD process.

#### 3.1.1 No-cloning principle

Starting from the peculiar behaviour of a qubit after measurement a fundamental principle has been defined: a qubit cannot be copied.

This principle, known as no-cloning principle, state that it is impossible to create an identical copy of an arbitrary quantum state without modifying it. This has been proved to be true either

if the qubit state is directly observed (with the consequence of destroy the original quantum state) and if the time evolution of the system is observed [23].

To prove it, consider that exists a unitary transform  $U$  that given a quantum state  $|\psi\rangle$  allows to copy this state on another one:

$$U |\psi 0\rangle = |\psi \psi\rangle \quad (3.1)$$

To get this result, expressing states as linear combination of  $|0\rangle$  and  $|1\rangle$ ,  $\psi$  and  $U$  should be related as follows:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad , \quad U |\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \quad (3.2)$$

Instead, apply the tensor product lead to the following result:

$$U |\psi\rangle = |\psi \psi\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \neq \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \quad (3.3)$$

The direct consequence of this principle is that a qubit sent between two parties cannot be copied to be analysed separately from the original. The only way for an eavesdropper to know the quantum state of a sent qubit is to measure it, with all the consequence this implies. Thus, in a quantum channel, it is possible to directly send the qubits representing a key, without hiding them with any mathematical operation.

This principle is the base of most quantum key distribution protocols such as BB84 and others belonging to the family of *prepare and measure* protocols.

No-cloning principle is sometimes confused with quantum teleportation. Quantum teleportation is a technique that actually allows to create a perfect copy of a qubit but, after this operation, the qubit it starts from is altered by required measurements.

### 3.1.2 Random Number Generators

As said, Shannon theory proves that a theoretical secure communication is possible by using one time pad only if three conditions are met: the key is of the same length as the message, the key (in whole or in part) is never reused and the key is truly random.

Quantum key distribution allows keys sharing by simply send qubits representing each bit of the key. This solves the first two requirements of Shannon theory, but bits belonging to the key must be selected beforehand to be encoded and sent. Hence, it is needed to ensure bits are truly random.

In classical computing generate a true random number is actually impossible because everything is deterministic. The only solution found is to simulate a true random generator behaviour with an algorithm that, starting from an initialization input seed is able to generate a random sequence. However, if the initialization input seed is known everyone can replicate the sequence. This kind of generators are known as pseudo random number generator (PRNG).

Pseudo random number generators cannot be used to generate a key since they do not meet the Shannon theory requirement. In classical computing random number generation problem has been solved by using additional devices that use a physical environment attribute that continuously changes to generate a true random number sequence.

In quantum computing random behaviour is intrinsic in quantum mechanics nature. In previous chapter it has been already proved that a qubit in superposition state, when measured, has 50% of probability to be measured in one of the two vectors composing the basis used for measuring. This means that a measurement of a qubit in superposition state has the same random probability as the toss of a coin.

Hence, true random numbers can be generated by exploiting the same quantum mechanic properties that allows the secure keys exchange. Thus quantum computing owns all the properties needed to construct a theoretical secure communication.

Random generators exploiting quantum phenomena are known as quantum random number generators (QRNG). Different implementations QRNG have been proposed, depending on the assumption that users trust the apparatus they use or not [24].

### 3.1.3 Quantum and classical channels

When qubit is sent from sender to receiver it is polarized in such a way that it represent a given information (typically a bit in classical computing). In order for the receiver to retrieve the correct information, he should measure the received qubit in the same basis as the sender to ensure he obtains the information the sender polarized for him. If he uses another basis, he gets a random outcome between one of the two vectors composing the basis and the original information is lost. It is therefore needed for sender and receiver to communicate outside the quantum channel, so they can agree on the basis to be used, to correctly measure the received qubit. Hence, quantum key distribution also requires a classical channel beside the already stated quantum one. Classical channel can also be used to carry additional information about the exchange. As example, sender and receiver can use known techniques to check that exchange actually took place without third party interferences.

Even if information about the basis used for the measurements are exchanged in a classical channel, this should not be a concern for security of the exchange providing that this information is sent after the exchange on quantum channel took place. In fact, since an eventual eavesdropper cannot make a copy of the exchanged qubits, the only way he has to steal information is making measurements while the qubits are in transit. In this scenario he can only try guessing basis for measurements. If, after that, information about basis to be used are disclosed, eavesdropper can no longer use this information and the communication is still considered safe.

Hence, classical channel for a quantum keys distribution purpose can also be a public channel. There is no need to encrypt information sent over this channel and anyone can read this information without representing a threat to the keys exchange security.

Sender and receiver should however be sure about the origin of the information they receive. In particular, if an eavesdropper measure qubits with his basis and then pretends to be the sender in the classical channel, he can transmit the set of basis he used and the receiver will believe that exchange went fine. Thus, the only important requirement for a classical channel is that it must be authenticated [25].

Quantum channels are the means used to carry quantum information. Usually they are classified by the bit-rate they can achieve or the distance they are able to cover. Quantum channels are a combination of devices used to polarize qubits, the means of transmission (cables/wires) and the end devices used to measure qubits. Usually current quantum channels make use of photons as qubits. Thus optical fiber is commonly used as mean of transport for qubit.

An important distinction for quantum channels regards the way they transmit information. In particular, they are known as Discrete Variable QKD channels (DV-QKD) if they are capable of generating and detecting very weak optical signals, such as a single photon for each information to be encoded. This is in contrast with Continuous Variable QKD channels (CV-QKD) that encodes information in signal waves such as light pulses.

CV-QKD allows to make use of conventional communication technologies. However DV-QKD grants higher transmission distance compared to CV-QKD and different kind of protocols have already been successfully implemented by exploiting this technology. This is the reason why in this work the attention will be focused only on DV-QKD devices.

In order to implement a discrete variable quantum key distribution sender should own a device capable of sending single photons. This is usually achieved employing laser pulses attenuated to single-photon level (weak laser pulse). True single photon sources are rarely employed mostly because of the limited emission rate this technology currently grants. The single photon these devices produce are used as quantum information carrier. The sender should send these photons to another device he owns, an encoder that, using a source of randomness to select encoding basis, encodes information on each photon. The kind of encoding must be selected depending on the features of both encoder of the source and detector of receiver. Information can be encoded in phase, polarization, arrival time (time-bin encoding [26]) or any other photon properties. Considering a system that encodes information on polarization, an encoder would be composed of a polarization filter, a foil, oriented in the axis representing chosen basis.

The final part of a quantum channel is the detector. When photons are received, receiver use a source of randomness to select basis for measurements as well as sender did on his side. In this case, receiver can use beam splitters to decode received information. After this process, photons

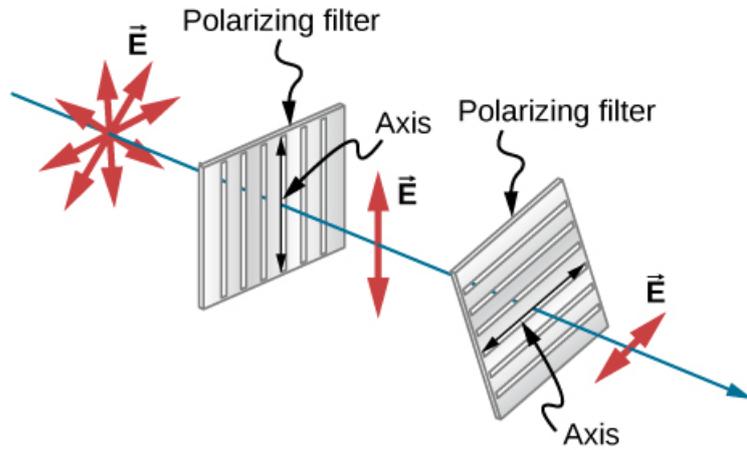


Figure 3.1. Example of polarization filter (source: [phys](#)).

arrive to detectors to complete measurements. In DV-QKD, receiver should employ single photon detectors, capable of transform an optical signal due to a single photon into a detectable electrical signal. When the photon is detected, the detector is said to ‘click’ in order to signal the received data. The most commonly used are InGaAs (Indium Gallium Arsenide) avalanche photodiodes capable of detect near-infrared light with high sensitivity and low noise [27].

A QKD system is composed of all these components together. The following schema represents a simple system employing real components:

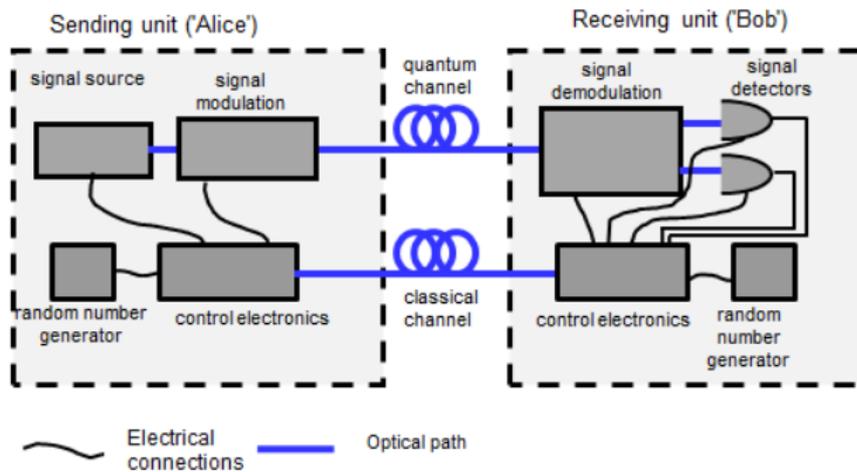


Figure 3.2. Simple QKD system (source: [ETSI GR QKD 003](#)).

### 3.2 First example with a noiseless channel

After having seen all components involved in a quantum key exchange, it is possible to imagine a practical implementation of these concepts to outline the main steps to be performed in a quantum key distribution. Consider two parties involved in the communication called Alice and Bob who will try to exchange a key. For this first example, all devices employed during the exchange are considered to be perfect devices: they will not introduce any error in the communication and everything will behave as expected.

In such a scenario the sender, Alice, will have to randomly chose bit belonging to the key. To this aim she will prepare a quantum circuit with a given number of qubits,  $n$ , and by using

a Hadamard gate will put them in superposition state. Now she can measure these qubit in the standard basis  $|0\rangle$  and  $|1\rangle$  to get a truly random sequence of bits.

Once key has been defined, Alice will proceed by encoding bits of the key in a set of qubits. She will choose (this can also be agreed beforehand with the receiver) a set of basis that can be used for the measurements and, by randomly picking one basis for each qubit, will encode the qubits in such a way that measuring them in the correct basis will return the generated key. When information are correctly encoded in qubits, Alice sends these qubits on the quantum channel and the receiver, Bob, stores them for subsequent measurements.

Once communication on quantum channel is over, Alice knows Bob has received the whole key and uses the classical channel to communicate the correct basis for the measurements. At this point Bob can measure its qubits using basis Alice communicated to him. Since all devices involved into communication are perfect and there is no one trying to eavesdrop the key, there is no interference and all qubit are correctly received. Thus measurements results will lead to the same key Alice sent.

When key transfer is over, Alice and Bob can apply some information reconciliation procedure to check if the exchange led to some errors that may reveal an eavesdropper interference. If everything is fine, Alice and Bob can use this key to securely encrypt a message.

### 3.3 Noise introduction and almost perfect security

The features explained so far lead to say that QKD can allow a theoretical secure communication as it meets all Shannon theory requirements. In an ideal scenario, QKD is effectively a perfect solution to the keys exchange problem. However, when it is used in reality, there are errors and problems that must be considered.

Besides errors that may be caused by a third party presence, when real devices are employed transmission errors may occur just as it happens for classical communication. Since real devices are not perfect, polarization errors, detection inefficiency, transmission lost as well as any external interference if the system is not well isolated can alter the state of a quantum bit. It is important to consider that this kind of issues may occur and foresee suitable mechanisms to detect and possibly correct these errors.

To take track of any kind of error that may occur during a communication over a quantum channel, the Quantum Bit Error Rate (QBER) has been introduced. These parameters reflects the percentage of errors that occur during a communication and it can refer both to channel errors as well as errors due to an eavesdropper. Since, potentially, every error on the channel is due to a third party trying to sniff the communication, it becomes important to be able to efficiently estimate QBER in order to understand the amount of information that has been leaked. Considering any kind of error source, QBER can be estimated as follows <sup>1</sup>:

$$QBER = p_f + \frac{p_d n q \sum f_r t_l}{2} \mu \quad (3.4)$$

Where

- $p_f$ : probability for a wrong ‘click’ of detector.
- $p_d$ : probability for a wrong photon signal.
- $n$ : number of detections.
- $q$ : phase =  $\frac{1}{2}$ ; polarization = 1.
- $\sum$ : detector efficiency.
- $f_r$ : pulse repeat frequency.

---

<sup>1</sup><https://www.quantiki.org/wiki/bb84-and-ekert91-protocols>

- $t_l$ : transmission rate (for large distances small).
- $\mu$ : attenuation for light pulses (single photons = 1).

If resulting QBER is too high, the communication is considered insecure and exchanged data should be discarded. QBER hence becomes an indicator of security for a quantum communication. In such scenario it is crucial to be able to identify the correct threshold for QBER: since channel errors too are included in QBER calculation, discarding communications that exceed a low threshold may lead to discard quite all communication; on the other hand selecting an high threshold may be risky if considering that potentially all errors are due to information leaks.

In 2000, Shor and Preskill presented a security proof study that identify the ideal QBER threshold for a quantum communication [28]. They demonstrated that with a QBER of 11% an eventual eavesdropper cannot recover exchanged information, even if all the errors are addressable to him. Hence it is common in quantum communication consider unsecure those information exchanged with a QBER above 11%.

### 3.3.1 Error correction problem

By sending known patterns of data through the quantum channel it is possible to make an estimation of the error introduced by the channel. Errors may occur in every part of the quantum channel. Probabilities of wrong polarization, light attenuation in the channel as well as erroneous click of detector may affect reliability of received data.

Quantum channel should be calibrated in order to reduce as much as possible QBER. A satisfactory QBER parameter for current quantum channels is about 1-2% of exchanged data. This means that, with current technology, a good quantum channel introduces about one or two errors over 100 photons sent. This unavoidable leads to the need of detecting and correcting communication errors.

In classical information error correction problem is mostly solved using backup information of transmitted data. The same data is made redundant in order to easily identify and correct any transmission error.

Quantum computing cannot make use of redundancy data because, as already seen with no-cloning principle, it is impossible to make a copy of a qubit to be transmitted. For long time, this has been considered a huge limitation in the realization of real quantum devices because of the lack of alternatives to copy for detecting and correcting errors. However, in 1995 Peter Shor presented a scheme for error detection and correction that does not violate no-cloning theory [29].

What Shor outlined is that, even if a qubit cannot be copied, its state can be stored into different entangled qubits. An error can interfere with the qubit by flipping it, changing its phase or the both of them. These operation corresponds to the Pauli matrices X, Y and Z. By storing the state of a qubit into nine different qubits is possible to perform a multi-qubit measurement that does not interfere with the qubit status, but discloses information about the kind of error occurred if any. Once the kind of error has been discovered it is possible to apply the correspondent Pauli gate to revert the effect while keep saving the original qubit status.

After this first scheme in 1995, a lot of new researches have been proposed, introducing a wide range of different error correction schemes. The most common is Cascade protocol [30]. Cascade protocol works on bits after they are exchanged over the quantum channel. Basically the two parties involved in the communication must divide exchanged information into blocks and check for errors by comparing parities of each block. If parity is different they have to perform a binary search to locate the error and must repeat these steps different times with different block size and permutation to make sure all errors are corrected.

Cascade protocol drawback is that it needs a lot of interaction over the classical channel. This is something that is better to avoid since classical channel is public. Forward error correction codes overcome this problem since they allow correcting errors by sending only one syndrome. A common scheme belonging to this family is Low Density Parity Check (LDPC) code [31].

### 3.3.2 Privacy amplification

As said, real quantum channels are subject to errors. Parties involved in the communication know that and therefore need to check keys after they are exchanged. Even if it is possible correct errors that may occur on some qubit, it is impossible to say if the error has been introduced by the channel or it is caused by the presence of a third party who tried to measure the qubits value. Due to this uncertainty, sender and receiver consider that any error that occurs is a piece of information that now belongs to a possible eavesdropper and thus they try to apply different countermeasures.

After error correction phase it is possible to reduce the knowledge the eavesdropper has about the key by applying privacy amplification.

Privacy amplification is a technique performed through classical channel to convert a weak secret into a uniform key that is fully secret from any eavesdropper. There are different kinds of privacy amplification schemes but most of them consist of reducing the length of the key in order to reduce the eavesdropper knowledge. This is usually done by employing hashing techniques on exchanged key. Applying hash functions to the key ensures the eavesdropper loses any knowledge of the key at the cost of reducing the length of the key.

Besides privacy amplification, it is optionally possible to use advantage distillation techniques. Since also channel errors are considered in QBER calculation, if a channel is particular noisy it is likely to exceed the 11% threshold in error rate. This can cause a lot of keys to be dropped with a consequent lowering of system key rate. To overcome this problem, advantage distillation techniques have been proposed [32]. These techniques allow to consider an higher QBER threshold before discarding the key as to increase produced keys rate of the overall system.

## 3.4 Quantum Key Distribution Protocols

With the rise of quantum technology different studies proposed many different implementations of quantum key distribution. Quite all presented protocols can be summarized into two different categories: “prepare and measure” and “entanglement based”.

Prepare and measure refers to those protocols that encode qubits information in given states and then expect the information to be retrieved by measuring qubits with the same basis they have been encoded. By exchanging information about the basis used during measurements, these kind of protocols can get information about an eventual eavesdropper presence and calculate the amount of information that has been stolen.

Entanglement based protocols exploit entanglement relationship between two or more qubits to carry information. As already explained, entanglement relationship is not tied to the distance the qubits are separated from hence, once this relationship has been established, the qubits can be sent from sender to receiver without losing their properties, regardless the distance of the two. Also in this case, measurements performed by a third party alter the system. Thus, for entanglement based protocols too, it is possible to detect a third party presence and calculate the amount of information he has been able to steal.

A list of currently developed QKD protocols is here reported:

- *BB84* [33].
- *E91* [34].
- *T12* [35].
- *Decoy state* [36].
- *SARG04* [37].
- *Six-state* [38].
- *B92* [39].

- *BBM92* [40].
- *MSZ96* [41].
- *COW* [42].
- *DPS* [43].
- *KMB09* [32].
- *HDQKD* [44].

The first quantum key distribution protocol is BB84. It was presented in 1984 and belongs to prepare and measure category. The first entangled based protocol is E91 and it has been introduced in 1991. During the years different improvements to these two protocols have been proposed leading to the above cited protocols. However, these two protocols remains the cornerstones of quantum key distribution and will now be described.

### 3.4.1 BB84

BB84 is the first quantum key distribution protocol developed. It is a prepare and measure protocol developed by Bennett and Brassard in 1984 [33] (the name of the protocol is a combination between the names of its inventors and publication year).

The protocol consists of a first phase where qubits are exchanged on a quantum channel and a second phase, run on the classical channel, where information about measurements are disclosed. Basically, the protocol leans on the no-cloning theory, hence qubits are prepared in a given state, representing a bit in classical computation, and they are sent over the quantum channel without worry about a possible eavesdropper.

Considering a practical case of a sender and a receiver, called again Alice and Bob, and an eavesdropper who tries to steal exchanged information, called Eve. To effectively understand reliability of QKD systems, consider that Eve has full access to both classical quantum channel as shown in figure 3.3.

The protocol requires to perform different steps. The first step consists of choosing a true random generated key and then encode the bits of the key in quantum bits by following a convention decided beforehand between the two parties. The basis used during polarization and detection phases as well as how they encode the classical bits are decided between the two parties depending on the devices capabilities they own. Usually it is enough to have two different basis that for simplicity are the rectilinear (+), which can polarize the qubit in vertical ( $0^\circ$ ) or horizontal ( $90^\circ$ ) way, and the diagonal basis (X), which lead to polarization of  $45^\circ$  and  $135^\circ$ . After having chose the basis a simple encoding can be proposed as follows:

Basis	0	1
+	↑	→
X	↗	↘

Table 3.1. Basis encoding.

Note that this encoding has been chosen just for example. The two parties can agree on the encoding they prefer as well as use completely different basis.

Once the sender, Alice, generated the key, she has to choose which basis use to encode each bit. This choice must again be performed randomly as to prevent anyone from knowing the basis used by using any other method than guessing. This ensures that, if the key is long enough, the probability an eavesdropper has to guess all the basis used and introduce no error is near to zero. Alice can now encode the bits of the key by polarizing the same number of qubits in such a way that, if the first bit of the key is 0, measure the first qubit with the basis Alice chose will always lead to 0 as result. If instead the other basis is used during measurement, the outcome

is completely random and can be either 0 or 1. The so polarized qubits are sent through the quantum channel directly to the receiver, Bob.

When Bob receives the qubits he immediately measures them. Qubits are not stored mostly because of the poor reliability in term of memory retain of current quantum devices. Thus Bob randomly chooses basis for measurements between the two possible basis allowed by the protocol and measure the qubits as soon as they are received.

When transmission over quantum channel ends, Alice and Bob communicate through the authenticated classical channel and disclose information about the basis used. If they used the same basis for a qubit, they expect to have the same corresponding bit providing that no one has interfered with the exchange. If instead they used a different basis, they cannot know if the measurement led to the right information, thus they consider that bit as invalid. After basis discussion over classical channel, Alice and Bob keep bits measured with the same basis and discard the ones measured with a different basis. In this way they obtain the same shared key. This procedure is known as key sifting.

All this process can be schematized by the following table:

Alice's random key	0	1	1	0	1	0	0	1
Alice's random basis	+	X	+	X	X	+	X	+
Alice's polarization	↑	↘	→	↗	↘	↑	↗	→
Bob's random basis	+	X	X	X	+	X	+	+
Bob's measurements	↑	↘	↘	↗	→	↗	↑	→
Basis discussion on classical channel								
Shared key	0	1	-	0	-	-	-	1

Table 3.2. Key sifting procedure.

After basis discussion the two parties get a common key string. However they still have no information about possible errors in the transmission. Last phase of the protocol consists of select a subset of the key string and compare it over the classical channel, in order to make an estimation of the QBER. The bits chosen for the QBER estimation are discarded by the final key since they can easily read by anyone on the public classical channel.

If the QBER is below a given threshold (11% [28]) the key is considered safe and the two parties can start using it, otherwise the eavesdropper is thought to know too much about the key that will therefore be discarded.

It can be easily seen how the presence of an eavesdropper increases the error rate. Starting from the above example of exchange an eavesdropper, called Eve for simplicity, can be introduced. Eve has no information about the basis to be used for measurements when qubits pass through the quantum channel, so she can only select them randomly. What follows can be schematized by the following table:

Alice's random key	0	1	1	0	1	0	0	1
Alice's random basis	+	X	+	X	X	+	X	+
Alice's polarization	↑	↘	→	↗	↘	↑	↗	→
Eve's random basis	+	+	X	X	+	X	+	X
Eve's polarization	↑	→	↘	↗	→	↗	↑	↗
Bob's random basis	+	X	X	X	+	X	+	+
Bob's measurements	↑	↗	↘	↗	→	↗	↑	↑
Basis discussion on classical channel								
Errors	0	x		0				x

Table 3.3. Key sifting procedure with eavesdropper presence.

Due to randomness of key and chosen basis, errors are considered to be uniformly distributed along the key, hence any subset proportional to the length of the key of the key should be enough to make a reliable estimation of the error rate.

If the error rate is below the security threshold, Alice and Bob have to proceed by correcting remaining errors in the key. They select an error correction scheme such as LDPC [31] and recursively apply it until all errors have been corrected.

After this step they can optionally apply privacy amplification techniques to reduce any information an eavesdropper may have. Alice randomly select an hash algorithm from a predefined set of allowed algorithms. She applies the hash function and communicate it to Bob over the classical channel so that he can perform the same operation on his key.

At this point the protocol ends and the so exchanged key can be used for cryptography purposes.

### 3.4.2 E91

E91 is a quantum key distribution protocol based on entanglement. Even this protocol gets its name by a combination of the name of its inventor and the year of publication since it was proposed by Artur Ekert in 1991 [34].

The protocol is quite simple. It consists of performing the following steps:

- Prepare qubit pairs in entanglement state.
- Send each qubit of the pairs to the two parties involved in the communication.
- Measure the received qubits: thanks to entanglement if qubits are measured in the same basis the same result will outcome.
- Verify entanglement on received qubits.

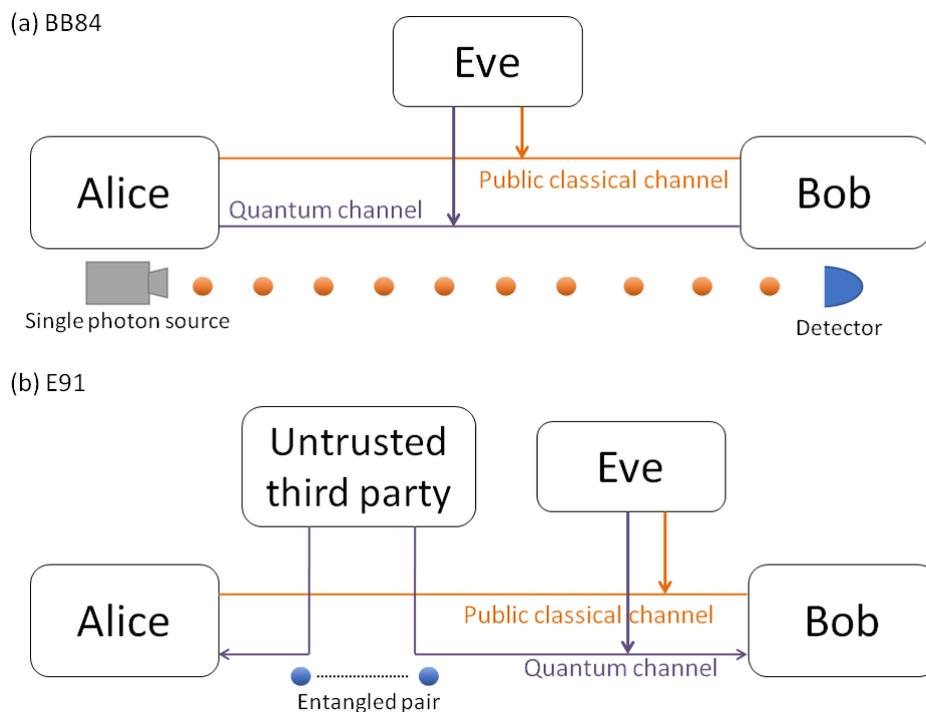


Figure 3.3. Conceptual schemes: fig (a) BB84 protocol; fig (b) E91 protocol.

To better clarify this protocol consider a practical case of a sender and a receiver, Alice and Bob. Alice can prepare entangled qubits and send one qubit of each pair to Bob. The two parties randomly choose basis for the measurements and perform measurement as soon as they get the qubits. After all qubits are received and measured and all its values are stored the two parties start a public discussion over the classical channel to identify the qubits they measured with the

same basis. For those qubits they know the outcome of their measurements is the same, hence they keep those result to construct the key and drop all other measurements.

The last step of this protocol consists of checking that the received qubits are actually entangled. This is done by calculating the correlation value of the received qubits, comparing it with CHSH inequality. In order to make sure that received qubits are in an entanglement state the correlation value  $C$  must satisfy the requirement:

$$C \simeq 2\sqrt{2} \tag{3.5}$$

This requirement is known as *Tsirelson's bound* and represents the maximal correlation value between two particles [45]. Hence a correlation approximately equals to this value can be considered the proof that no one interfered with the communication. If instead someone interferes with the communication, CHSH inequality is not violated and the correlation value will be in the range:

$$-2 \leq C \leq 2 \tag{3.6}$$

indicating that an interference happened.

A hypothetical eavesdropper, Eve, can intercept also both of the entangled qubits and measure them to obtains the results. However, since measurements destroy qubit state she will need to prepare two new qubits for Alice and Bob. She can encode in these qubits the measurement results she got and, if Alice and Bob chose the same basis, all the parties will get the same results. However the qubits sent by the eavesdropper are not entangled anymore and the check on the correlation value will let Alice and Bob realize that someone interfered with the communication. After that Alice and Bob discard the key.

It is important to notice that, thanks to entanglement properties, it does not matter who prepares the entangled qubits. The source of qubits can also be an untrusted party or, even worse, the eavesdropper itself. This does not affect the security of this protocol since whoever prepares the qubits cannot measure them as explained above. This feature is known as *Device Independent QKD* (DI-QKD), since the security of the protocol is granted even when untrusted QKD devices are employed [46].

# Chapter 4

## QKD standard

### 4.1 ETSI GS QKD

With the increasing interest in quantum key distribution, different entities try to standardize procedures and operations to correctly perform key exchange. A standard to follow becomes crucial when different companies start to develop devices or applications based on QKD. It ensures applications can interoperate with devices coming from different companies, assuring the same minimal functional requirements and security levels.

In this work attention will be focused on quantum key distribution standard defined by the *European Telecommunications Standard Institute* (ETSI). ETSI is a no-profit standardization authority that produces globally applicable standards for ICT-enabled systems <sup>1</sup>. Regarding QKD, ETSI Industry Specification Group (ISG) proposed a set of documents, called Group Specification (GS) to specify QKD system interfaces, implementation security requirements and optical characterization of QKD systems and their components. Each one of these GS documents is aimed to standardize a different aspect of QKD, from hardware interfaces to software applications.

ETSI standard is composed of the following documents:

- *Vocabulary (ETSI GS QKD 007)* [47]. This standard collects definitions and abbreviations related to QKD and used in the other documents. It is an introductory document to clarify terminology in order to reduce confusion for readers.
- *Device and Communication Channel Parameters for QKD Deployment (ETSI GS QKD 012)* [48]. This document describes the communication resources involved in a QKD system. It is mainly focused on fibre optical networks and point-to-point communication. Besides, it also defines QKD architectures intended as the possible design choices that lead to specify how each communication requirement is accommodated.
- *Components and Internal Interfaces (ETSI GS QKD 003)* [49]. It describes properties of hardware components involved in QKD process. In particular, it describes photon sources and detectors for both DV-QKD and CV-QKD outlining the main parameters that characterize them.
- *Component characterization: characterizing optical components for QKD systems (ETSI GS QKD 011)* [50]. This document is intended to specify procedures for their characterization. It defines procedures to measure optical sources and detectors properties since a misconfiguration of one of these components may affect overall security.

---

<sup>1</sup><https://www.etsi.org/>

- *Security Proofs (ETSI GS QKD 005)* [51]. It is a reference document for the construction of requirements and evaluation criteria for practical security evaluation of QKD systems. It identifies the critical characterizations of the commonly used QKD components, clarifying the difference between the security claim of a protocol, based on models, and the security claim of its implementation.
- *Use Cases (ETSI GS QKD 002)* [52]. Use Cases lists a set of possible application scenarios in which QKD systems may be employed as building blocks for high security information systems. The final goal of this document is to present specifications and mechanisms for driving development towards a security certification of QKD systems.
- *Application Interface (ETSI GS QKD 004)* [53]. This is the document that specifies the Application Program Interface (API) that high level application should use to interact with QKD implementations compliant to ETSI standard. This document will be further investigated in the next section.
- *Protocol and data format of REST-based key delivery API (ETSI GS QKD 014)* [54]. This document specifies the REST-based interface that allows high level application to communicate with standard compliant device over a network. Also this document will be investigated below.

#### 4.1.1 Application Program Interface

Application Interface [53] is the document that specifies the Application Program Interface (API) between the QKD Key Manager and applications. A Key Manager is in charge to manage secure keys produced by an implementation of a QKD protocol and to deliver, on demand, identical sets of keys, via this API, to peer applications at the communication end points. Manufacturers may adopt this API in order to make their products interoperable with versions provided by other vendors. API defined here refers to local key manager specification. ETSI specified another API that can be built on top of this. This is the REST API defined in ETSI GS QKD 014 that will be discussed later on in this paragraph.

The underlying design is to have an API able to create a continuous stream of keys from two (or more) connected remote applications in order to maximize key throughput with a minimal overhead. A simple example can be seen in figure 4.1. This figure shows a single QKD link, with end points at Site A and Site B. Each site has a single application (shaded in yellow) and a single QKD module, enclosed by a blue box, which implements its part in a QKD protocol (shaded in red) to produce QKD keys that are managed by a QKD key manager peer (shaded in green). The single peer application in this case, uses the QKD application interface to acquire identical sets of secure keys in the two sites on demand.

This API is designed to be as small as possible while being highly flexible. The defined application interface consists of three functions, OPEN\_CONNECT, GET\_KEY and CLOSE, whose syntax is shown by figure 4.2.

**OPEN\_CONNECT()** function reserve an association for a set of future keys at both ends of the QKD link. This association is identified by `Key_stream_ID` parameter. It is a blocking function that returns only when connection with target key manager is established or when a specified timeout occurs. Timeout, as well as other settings, can be specified through `QOS` parameter to define the expected levels of key service (Quality Of Service). More specifically, `QOS` is a structure collecting parameters as specified by table 4.1.

`Key_stream_ID` is a unique identifier for the group of keys provided by the key manager to the application. It shall be in the form of a UUID version 4 <sup>2</sup> and, although it is a reference to locate a key, it should not contain any key material (i.e. key, in whole or in part, cannot be derived from `Key_stream_ID`). Both peers must use the same `Key_stream_ID` in order to recover the same key from their key modules, hence it can be shared on the public channel and stored by peer applications. **OPEN\_CONNECT()** can be called with `Key_stream_ID` set to `NULL` to

---

<sup>2</sup><https://tools.ietf.org/html/rfc4122>

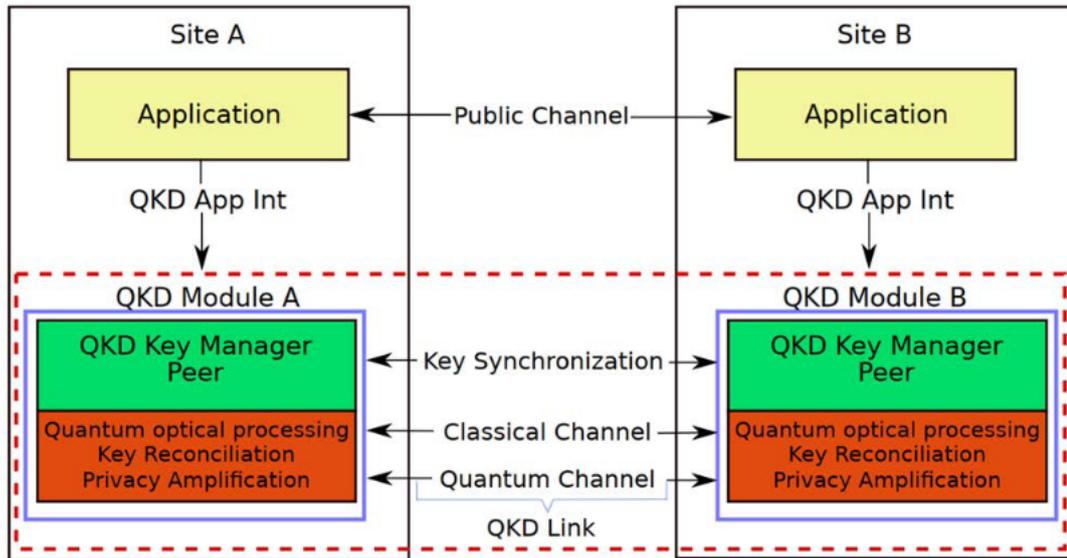


Figure 4.1. QKD Application Interface and peer relationships (source: ETSI GS QKD 004 [53]).

```

Interface QKD{
    OPEN_CONNECT (in source, in destination, inout QOS, inout Key_stream_ID,
                 out status);
    GET_KEY (in Key_stream_ID, inout index, out Key_buffer, inout Metadata, out
            status);
    CLOSE (in Key_stream_ID, out status);
}
    
```

Figure 4.2. ETSI Key Manager application interface.

let key manager choose an appropriate ID, in this case the assigned ID will be returned by `OPEN_CONNECT()` function itself and the calling application will be in charge of sharing this ID with the other peer of the communication (how this is done is outside the scope of the standard). Otherwise the function can be called with a given ID, that can be either an ID received by the peer end who recently used this function or a previously established ID. An application may require and manage several independent `Key_stream.ID` without limitations.

Source and destination parameters are both URI, identifying the source application and the destination application respectively.

Status is instead an output parameter whose values are described by table 4.2.

**GET\_KEY()** is the function used to retrieve a key from the key manager. It returns a key associated to the `Key_stream.ID` passed as input parameter together with the index specifying the position of the key to be retrieved inside the stream. Besides, if function is called with `Metadata` parameter different from zero the function will return additional information about the key chunk. If it is zero, no metadata will be returned.

This function may be called as often as desired, but the key manager only needs to respond at the bit rate requested through the `QOS` parameters, or at the best rate the system can manage. The QKD key manager is responsible for reserving and synchronizing the keys at the two ends of the QKD link through communication with its peers. Success or failure of this function can be checked by looking at returned status parameter, whose meaning is described by table 4.2.

**CLOSE()** function terminates the association established for the `Key_stream.ID` passed as input parameter. Also for this function the output status parameter will show whether the

QOS		
Field	Description	Type
Key_chunk_size	Length of the key buffer, in Bytes, requested by the application	32 bit unsigned integer
Max_bps	Maximum key rate, in bps, requested by the application	32 bit unsigned integer
Min_bps	Minimum key rate, in bps, required by the application	32 bit unsigned integer
Jitter	Maximum expected deviation, in bps, for key delivery	32 bit unsigned integer
Priority	Priority of the request	32 bit unsigned integer
Timeout	Time, in milliseconds, after which the call will be aborted, returning an error	32 bit unsigned integer
Time to Live (TTL)	Time, in seconds, after which the keys for this key stream ID shall be erased from the dedicated key store of the application, for security reasons	32 bit unsigned integer
Metadata	The mimetype of the metadata to be delivered by the Key Manager on each GET call	char array of size 256

Table 4.1. QOS structure fields.

Status	
Value	Meaning
0	Successful
1	Successful connection, but peer not connected
2	GET_KEY failed because insufficient key available
3	GET_KEY failed because peer application is not yet connected
4	No QKD connection available
5	OPEN.CONNECT failed because the Key_stream_ID is already in use
6	TIMEOUT_ERROR The call failed because the specified TIMEOUT
7	OPEN failed because requested QoS settings could not be met
8	GET_KEY failed because metadata field size insufficient. Returned Metadata_size value holds minimum needed size of metadata

Table 4.2. Status parameter meaning.

function executed correctly.

Beside the defined interface to interact with high level application, standard does not define internal requirements for key manager. Specifically it does not require the interface to be build with a particular programming language, library or tool, neither how to manage internal operations or quality of service. It is up to implementer selecting implementation details to better satisfy the service requirements.

The standard also reports a sequence diagram showing the workflow that two applications, APPA and APPB, perform in order to retrieve a shared key (figure 4.3)

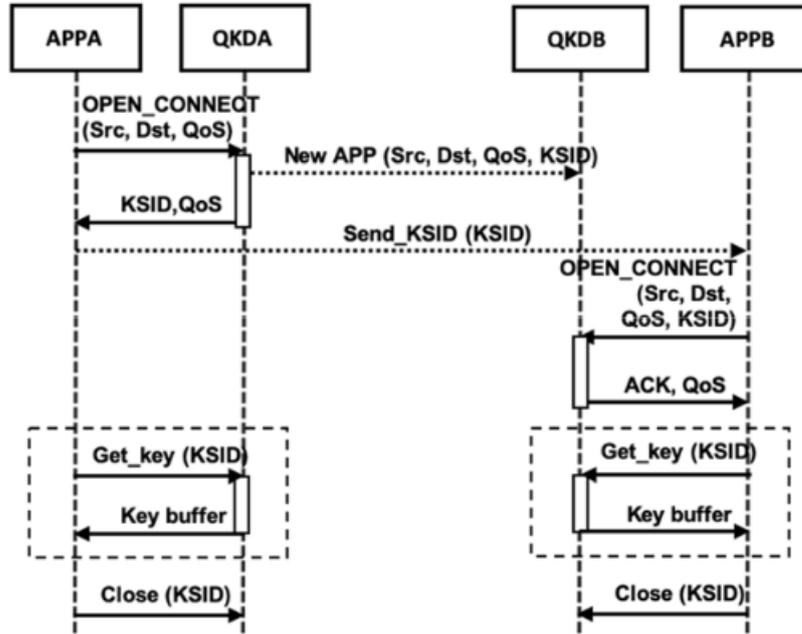


Figure 4.3. Simple example of two applications requiring a key. (source: ETSI GS QKD 004 [53]).

The image shows APPA starting the sequence with a call to OPEN\_CONNECT() indicating application source, destination, the desired QoS and a NULL value in the Key\_stream\_ID parameter. Key manager select an appropriate Key\_stream\_ID and forward it to its peer, then it returns the Key\_stream\_ID and status value to the caller application. APPA sends this Key\_stream\_ID value to APPB on the public channel who, on its side, will call OPEN\_CONNECT() to register the same ID as soon as it is received. At this point, if the call to OPEN\_CONNECT() function is successful for both of the APP, the key association has been verified and established. After that, both APPA and APPB start making calls to GET\_KEY() until they have no more need for secure keys and call the CLOSE() function in order to terminate the Key\_stream\_ID association. How both key manager and high level application forward the Key\_stream\_ID to end peers (functions New\_APP() and Send\_KSID()), it is not defined by the standard and can be implemented as desired.

#### 4.1.2 REST-based interface

The Group Specification 014 [54] describes the REST-based interface for a quantum key distribution network. This standard introduces the following notation for the entities involved in the communication: Key Management Entity (KME) is the entity that manages keys in a network in cooperation with one or more other Key Management Entities; Secure Application Entity (SAE) is the entity that requests one or more keys from a Key Management Entity; QKD Entity (QKDE) is the entity providing key distribution functionality (e.g. a QKD Module - key manager implementing an underlying QKD protocol - as defined in GS QKD 004). All SAEs and KMEs must possess an identifier that allows to uniquely identify them in the network.

Whilst Application Interface standard does not impose any implementation details, REST-based interface requires the use of HTTPS protocol and JSON data format. Standard also makes some security assumptions about entities and their relationship. In particular: each Trusted Node is securely operated and managed; this API is used between SAEs and KMEs within a secure site; each SAE is secure; each KME is secure.

REST API consists of three methods as depicted in table 4.3

**Get status** method returns Status from a KME to the calling SAE. Status contains information on keys available to be requested by a master SAE (the caller) for a specified slave SAE (the

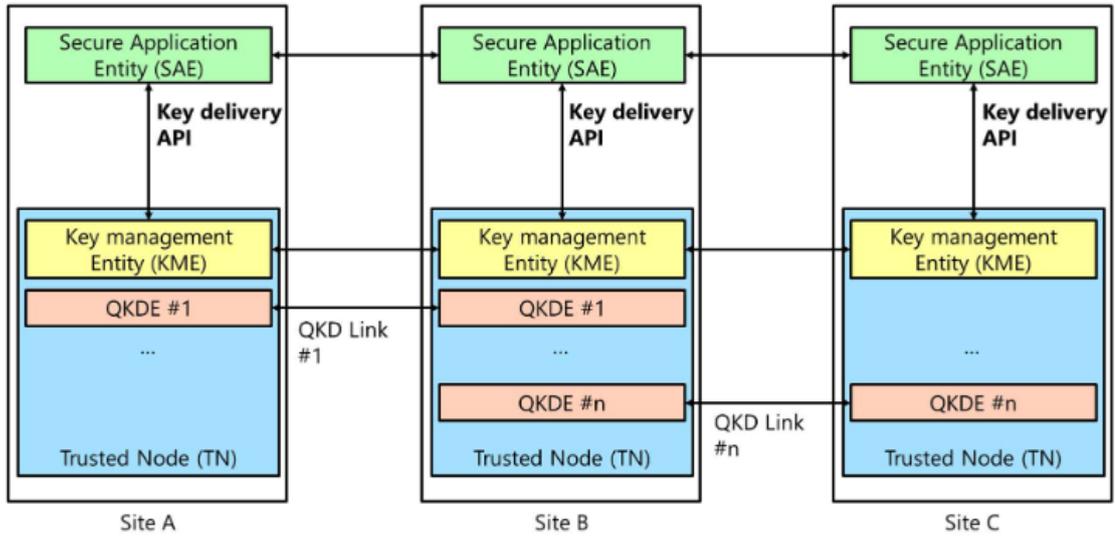


Figure 4.4. Example of QKD network. (source: ETSI GS QKD 014 [54]).

REST-based interface		
Method Name	URL	Access Method
Get status	https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/status	GET
Get key	https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/enc_keys	POST (or GET)
Get key with key IDs	https://{KME_hostname}/api/v1/keys/{master_SAE_ID}/dec_keys	POST (or GET)

Table 4.3.

target). It can be called with a GET request. As per REST specification, GET request does not contain a body with additional parameters. Hence the access URL should specify all parameters needed to satisfy the request. The access URL is the following:

`https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/status`

where `KME_hostname` is the hostname or IP address to which request is addressed and `slave_SAE_ID` is the URL-encoded SAE ID of slave SAE.

Response to this call is a JSON data format containing the following parameters:

- *source\_KME\_ID*: a string representing the KME ID of the KME the request was addressed to.
- *target\_KME\_ID*: a string representing the KME ID of the target KME (the one connected to the slave SAE).
- *master\_SAE\_ID*: a string representing the SAE ID of the calling master SAE.
- *slave\_SAE\_ID*: a string representing the SAE ID of the specified slave SAE.
- *key\_size*: an integer representing the default size of key the KME can deliver to the SAE (in bit).
- *stored\_key\_count*: an integer representing the number of stored keys KME can deliver to the SAE.
- *max\_key\_count*: an integer representing the maximum number of stored\_key\_count.

- *max\_key\_per\_request*: an integer representing the maximum number of keys per request.
- *max\_key\_size*: an integer representing the maximum size of key the KME can deliver to the SAE (in bit).
- *min\_key\_size*: an integer representing the minimum size of key the KME can deliver to the SAE (in bit).
- *max\_SAE\_ID\_count*: an integer representing the maximum number of additional slave SAE IDs the KME allows. If “0” the KME does not support key multicast.
- *status\_extension*: an object specified for future use.

**Get key** method returns Key container data from the KME to the calling master SAE. Key container data contains one or more keys. The calling master SAE may supply Key request data to specify the requirement on Key container data. The slave SAE specified by the *slave\_SAE\_ID* parameter may subsequently request matching keys from a remote KME using *key\_ID* identifiers from the returned Key container. This method can be called with POST request. Optionally GET request is allowed only if there is no need of optional parameters in the request body besides key number and/or size. In this case number and size can be passed as query parameters in the request URL. The access URL for this method is the following:

`https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/enc_keys`

where *KME\_hostname* and *slave\_sae\_ID* have the same meaning as the Get status request. Request body (for POST requests only) is represented by the JSON object Key request, which contains the following fields:

- *number*: an integer representing the number of keys requested, default value is 1.
- *size*: an integer representing the size of each key in bits. If not specified, default value is defined as *key\_size* in Status data format.
- *additional\_slave\_SAE\_IDs*: an array of strings used for specifying two or more slave SAEs to share identical keys. The maximum number of IDs is defined as *max\_SAE\_ID\_count* in Status data format.
- *extension\_mandatory*: an array of extension parameters specified as name/value pairs that KME shall handle or return an error. Parameter values may be of any type, including objects.
- *extension\_optional*: an array of extension parameters specified as name/value pairs that KME may also ignore.

Response to this request is given by the JSON object Key container defined by the following key-value pairs:

- *Keys*: an array of objects specified as follows:
  - *key\_ID*: a string representing the ID of the key in UUID format.
  - *key\_ID\_extension*: an object specified for future use.
  - *key*: a string representing the key data encoded by base64.
  - *key\_extension*: an object specified for future use.
- *key\_container\_extension*: an object specified for future use.

**Get key with key IDs** method returns Key container from the KME to the calling slave SAE. Key container contains keys matching those previously delivered to a remote master SAE based on the Key IDs supplied from the remote master SAE in response to its call to Get key. This method can be called with POST or GET request by following the same restrictions as Get key method. Specifically, GET request is allowed only when a single key ID needs to be requested and no extension is specified. In this case key ID is passed as query parameter of GET request. The access URL is the following:

`https://{KME_hostname}/api/v1/keys/{master_SAE_ID}/dec_keys`

where `KME_hostname` is the hostname or IP address to which request is addressed and `master_SAE_ID` is the URL-encoded SAE ID of the remote master SAE who requested keys with Get key. The request body (for POST requests only) is simpler than Get key request. In this case it is enough to specify a Key IDs JSON object composed of the following key-value pairs:

- *Key\_IDs*: an array of objects specified as follows:
  - *key\_ID*: a string representing the ID of the key in UUID format.
  - *key\_ID\_extension*: an object specified for future use.
- *key\_IDs\_extension*: an object specified for future use.

Response to this request is given by the same JSON object Key container already described for Get key request.

REST-based standard can be considered an easy implementation of API defined in GS QKD 004. In particular, Get key (as well as Get Key with key IDs) can be mapped to the three function `QKD_OPEN()`, `QKD_GET_KEY()` and `QKD_CLOSE()`, considering that `key_ID` is used as `Key_stream_ID` described for those functions.

## Chapter 5

# QKD criticalities and known attacks

Quantum key distribution claims to be the solution to achieve a fully secure communication. The reasons behind this assumption have been widely discussed in previous chapters and, theoretically, proved. However, different studies of the subject outlined possible criticalities that may affect the security of QKD systems and hence require attention during design phase. These criticalities lead to different kind of attacks that will be discussed in this chapter.

### 5.1 Classical channel

Up today, all developed quantum key distribution protocols still need a classical channel in order to correctly elaborate information exchanged over quantum channel. As previously said, the classical channel can be a public channel, allowing anyone to access the information there exchanged. The only requirement needed to be satisfied to ensure protocols security is that it must be authenticated [25] in order to avoid a classical man in the middle attack.

Authentication is the process of making sure a subject is really who it claims to be. Since in QKD an eavesdropper is considered to have full access to both quantum and classical channel it becomes crucial for the parties involved in the communication to make sure information comes from a trusted source. If this requirement is not satisfied a man in the middle can modify information exchanged on classical channel without leave any trace of its intervention. In this case the two parties cannot realize whether the communication over quantum channel was manipulated by a third party presence.

For a QKD classical channel concept of authentication includes also integrity. Integrity regards making sure data exchanged have not been modified in the path between sender and receiver. Since, with currently used authentication schemes, authentication is usually achieved by signing an hash of the data sent, integrity is granted by the same authentication procedure.

In a classical channel authentication, such as integrity, can be achieved with an asymmetric encryption algorithm on the data exchanged. Basically, sender signs any information sent on the classical channel with its private key. Receiver can verify provenance of these data, as well as their integrity, by checking their signature with sender public key. If data change during the transmission the signature verification procedure fails. However, currently used asymmetric encryption suffers from the problems already described in previous chapters: it can be broken by the advent of quantum computers by means of Shor's algorithm [10].

This is one of the most critical points in QKD development. To avoid any kind of attack that may occur on classical channel old asymmetric encryption should be dropped and authentication scheme based on quantum cryptography should be adopted. When this is not possible, post quantum cryptography algorithms should be employed to avoid entrust security to schemes that have already proven to be broken. However it is important to notice that authentication needs to

be valid only during key exchange phase. If someone breaks authentication after key exchange is complete the key can still be considered safe since no information about the key was exchanged over the public classical channel.

## 5.2 Repeaters

Repeaters in quantum key distribution have been introduced to overcome limits of current quantum technology that allows transmission only within restricted distances before incurring in losses. However repeaters are fundamental devices in order to be able to construct a shared quantum network (figure 5.1).

In order to share a key over a distributed network current solution employs trusted repeaters. If two parties need to exchange a key over long distance, a trusted repeater is put in the middle between them. The idea is to forward keys in a hop-by-hop fashion: the sender exchange a key with the repeater that then proceeds by exchanging the same key with the receiver. If the distance between the two parties requires it, more than one repeater can be employed exploiting the same mechanism [55].

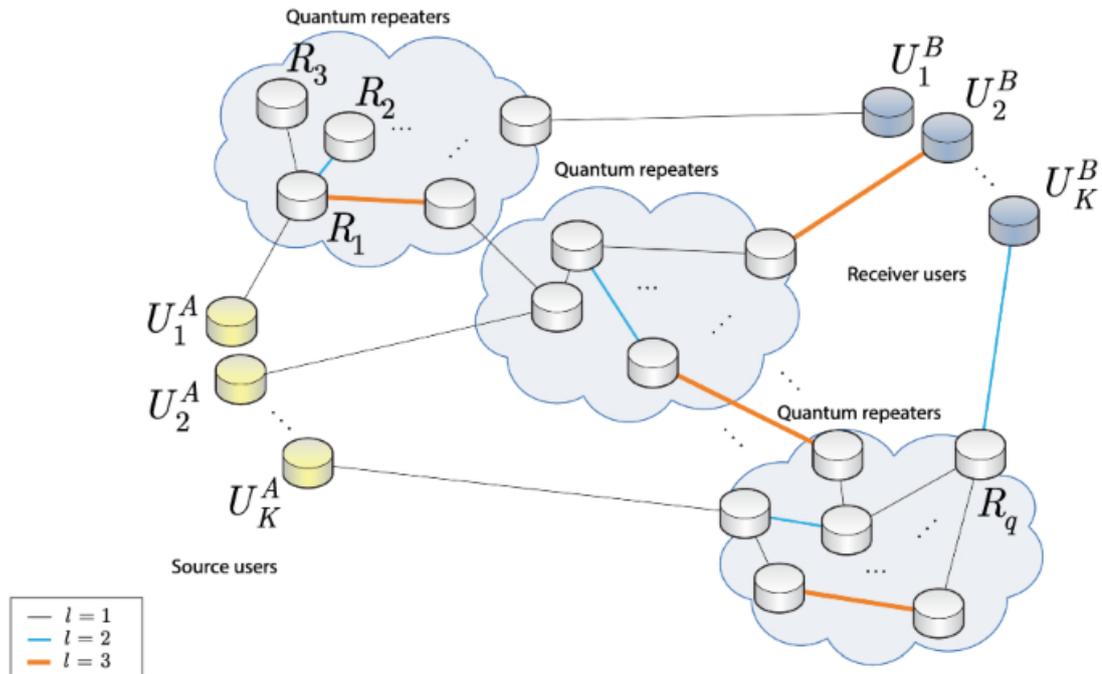


Figure 5.1. Example of quantum network. (source: ResearchGate).

This scenario implies that the key is decoded in clear inside each repeater it passes through. This is the reason why repeater must be trusted. Using an untrusted repeater would allow an adversary to gain knowledge about the key: if he owns the repeater he can get access to the whole key.

The need to relay on devices owned by other is a huge criticality. Security built on top of quantum mechanics properties loses its meaning if the overall security is entrusted to the assumption that no one in one of the repeaters will use keys for their own purposes.

A solution to the problem introduced by trusted repeater is to use *quantum repeaters* [56]. Among the various techniques, the most developed quantum repeaters exploit a phenomenon called *entanglement swapping* [57]. A schema of entanglement swapping can be seen in figure 5.2 in which ions particles (positively or negatively charged atoms) are subject to this process. To realize entanglement swapping two pairs of entangled particles, A - B and C - D are sent to the repeater.

The repeater task is to take one particle from each pair, B and C, and create entanglement between them. This will cause entanglement relationship to break on the original particles but it will be created on the two particles not processed by the repeater, A and D. At this point the repeater can send A and D particles to each one of the two parties in the communication that can now process the two entangled particles without release information during the exchange.

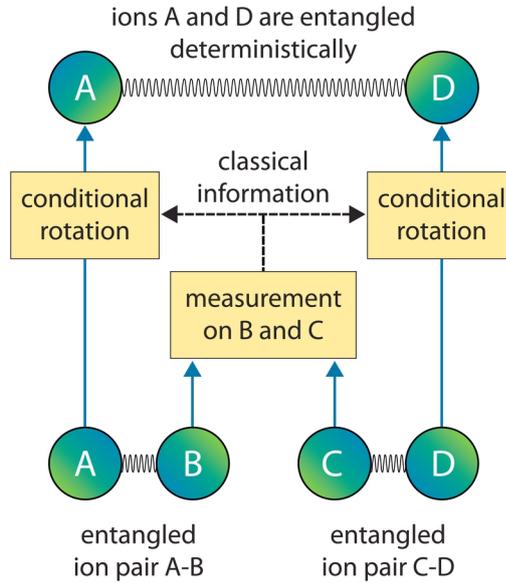


Figure 5.2. Entanglement swapping. (source: [phys.org](https://phys.org)).

### 5.3 Time shift attack

Time shift attack, such as all the following attacks explained in this chapter, exploits weaknesses and peculiar features of devices used in a QKD system. A real device may introduce errors and interference that are usually not taken into account in a simulated environment.

Time shift attack may take place when single-photon detectors in gated mode are used during QKD process. A single photon detector is a special kind of detector with a so high level of sensitivity that allows to count any single photon entering the device. This device is particularly useful to implement photons counting. For this kind of devices, the following parameters must be taken into account:

- Dark count rate: this parameter takes into account the average false detection of photons that may be caused by different reasons. The best detectors have a low dark count rate. Unfortunately, trying to reduce this parameter, leads to the time shift attacks.
- Dead time: it is used to take into account the time before a new measurement can be made after another one is just completed. During this time, detector will not be able to count any new photon.
- Quantum efficiency: it represents the fraction of incident photons which can be registered. A small quantum efficiency causes the photons to be not correctly detected.
- Timing jitter: this regards the uncertainty of the timing of the registered photon events.

Most single photon detector are realized with thermoelectric photo diodes. Heating in this kind of detectors is one of the main cause of high dark count rate. To minimize the dark count rate cooled diodes can be used, but it is also possible to use detector in gated mode: it is activated

only for a narrow window when a photon is expected to arrive. All photons arriving outside this window are not counted by detector. In such scenario, synchronization between source and detector becomes crucial.

In most applications two detectors are used to receive quantum states: one will detect bit “0” and the other one bit “1”. Even if the same signal is used to synchronize the detectors, some electrical imperfection and other factors may cause a different time response of the devices. This is solved by using a detection window much greater than the laser pulse duration. If timing mismatch between the two device is relevant, detection efficiency ( $\eta$ ) is different for the two devices and there may be some moments in which just one of the two detectors is enabled: consider  $t_1$  as the instant of time in which detector 0 is active while activation of detector 1 is still in progress, at the same way  $t_2$  is the moment in which detector 0 has come back off while detector 1 is still active. The same conditions can be found if just one detector is used for either “0” and “1”. In this case the expected arrival time for bit “0” ( $t_1$ ) is different from the one for bit “1” ( $t_2$ ), with the time difference determined by the optical length difference.

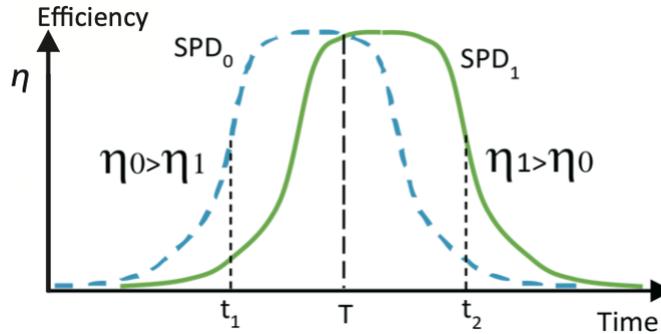


Figure 5.3. Time-dependence efficiency of Single Photon Detectors (SPDs) [58].

An attacker can just intercept the signal and, without measure it, shifts the time of the quantum state to make sure it arrives at time  $t_1$  or  $t_2$  at his choice. In this way, whenever receiver detects a signal, attacker knows the signal is a 0, if he chose  $t_1$ , or a 1 if he chose  $t_2$ . It is worth to notice that, since attacker has not measured the quantum signals, sender and receiver have no way to understand the distributed key has been tampered by using the classical checks of BB84 protocol.

In the event of a measure made with 2 detectors, the attacker can get full information about the key only if he finds an instant in which the two detectors are not active simultaneously. If this does not happen, attacker cannot know which one of the two detectors clicked. Besides, sender and receiver can realize the presence of an eavesdropper by analysing the detection efficiency of receiver that will be reduced by the presence of the attacker.

Presence of an attacker can also be detected if receiver uses a single detector. In this case, it is enough to choose a detector with a large gating period, enough to monitor pulses falling outside the predetermined windows for “0” and “1” [59].

Knowing the main flaws of this kind of detectors, it is possible to develop the right countermeasures to restore the expected security of the protocol. Proposed solutions regard taking into account the difference between the response curves of the detectors and removes the leaked information through privacy amplification techniques. Besides, it is also possible to randomly switch the bit assignment between the two detectors, preventing eavesdropper to assign bit values to the information he retrieved. [60].

## 5.4 Photon number splitting attack

Just like the previous category of attacks, photon number splitting attack takes advantage of flaws introduced by real device used during QKD.

In BB84 protocol any measure on the photon representing the quantum state alter its state and the two parties of the communication can be aware of the eavesdropper presence by monitoring the quantum bit error level (QBER). This is true as long as a single photon is used to represent the quantum state. Some experimental devices capable of emitting single photons have already been developed, but the photons emission rate is still limited and they require operational conditions not easy to meet. To realize QKD with currently available technology a simple, low cost and easy to integrate solution is using laser pulses attenuated to a very low level. This kind of devices are capable of emitting a number  $n$  of photons per pulse with a probability described by the Poisson distribution [61]. This means that, for each quantum state that needs to be encoded, one or more photons representing the same quantum state will be emitted.

This configuration opens an important security breach known as photon number splitting attack: an eavesdropper can perform a quantum non-demolition measurement that allows to measure the number of photons without disturbing the quantum states. Eavesdropper can block all pulses with a number of photons equals to one whilst, if the number of photons is greater than one, he can use a beam splitter (or any other suitable device) to separate the photons: one photon will be saved by him, the other ones will be regularly sent to the receiver. During the public basis discussion, the attacker can get information about the basis used to measure a given quantum state and use that basis to measure the copy he has stored. In this way the attacker can obtain the same key of sender and receiver without modify the QBER value, thus without revealing his presence.

To make the system tolerant to this kind of attack different solutions have been proposed.

One possible solution consist of a modified version of BB84 protocol, known as SARG04 protocol [37]. In this protocol sender encodes two strings of bits,  $a$  and  $b$ , in quantum states, where the  $i$ -th bit of the string  $(a_i b_i)$  can be encoded in one of these possible states:

$$\begin{aligned}
 |\psi_{00}\rangle &= |0\rangle \\
 |\psi_{10}\rangle &= |1\rangle \\
 |\psi_{01}\rangle &= |+\rangle = \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle \\
 |\psi_{11}\rangle &= |-\rangle = \frac{1}{\sqrt{2}} |0\rangle - \frac{1}{\sqrt{2}} |1\rangle
 \end{aligned} \tag{5.1}$$

$b_i$  represents the basis in which  $a_i$  is encoded (computational or Hadamard). Receiver, such as any possible eavesdropper, will choose a basis for each bit to perform measurements. At this point, instead of sharing the basis choice, sender sends for each bit a couple of possible state on the classical channel. Ad example, if the  $i$ -th bit is encoded as  $|\psi_{00}\rangle$ , thus  $a_i = 0$  and  $b_i = 0$ , sender will state on the public channel the two states  $|\psi_{00}\rangle$  and  $|\psi_{01}\rangle$  which represents all possible states with  $a_i = 0$ . At this point, receiver, who has already measured the quantum state sent with a randomly chosen basis, will compare his result with the two states sent by sender: since sender sent  $|\psi_{00}\rangle$  state, if receiver measured this state with the computational basis he will obtain  $|\psi_{00}\rangle$  and, since it is one of the possible state, he cannot gain information about what was the real state sent. In this case receiver states that this bit is invalid and it is skipped from the final key. If, instead receiver chose Hadamard basis to make his measurement, two possible results may outcome:  $|\psi_{01}\rangle$  and  $|\psi_{11}\rangle$ . Again, if result is  $|\psi_{01}\rangle$ , since it is one of the possible states announced by sender, nothing can be inferenced and the bit is declared invalid by receiver. If result of measurement leads to  $|\psi_{11}\rangle$  receiver can notice that he choose the wrong basis for measurement, hence the expected basis was computational one and the actual state sent from sender was  $|\psi_{00}\rangle$ . Receiver can now inform sender that  $i$ -th bit is valid and bit  $b_i$  can be added to the key. In such a scenario, in order to get information about the exchange key, eavesdropper would need more copies of the same qubit in order to perform all the measurements needed in both possible basis to get reliable information about the exchanged bit.

Besides resistance of this protocol to photon number splitting attack, it has been found vulnerable to other kinds of attacks [62], hence other solution has been developed.

Most used solution for photon number splitting attack is known as decoy state [36]. It basically consists of sending the signal state and several decoy states using different level of intensity.

Varying intensity leads to vary the photon number statistics through the channel. Thus, each intensity level has a QBER associated with it. During public discussion over classical channel, sender can disclose information about intensity level chosen for the transmission of each qubit. In this way, by monitoring QBER value associated at each intensity level, it is possible to understand if a photon number splitting attack has occurred.

## 5.5 Man In The Middle (MITM)

When talked about quantum protocols it was already stated that any attempt for an eavesdropper to intercept, measure and resend information on the quantum channel will be recognized by the two parties involved in the QKD process. MITM attack in a quantum device is therefore not related to the key exchange itself, but to a process that take place before the key distribution starts: calibration of quantum channel.

Almost all QKD detectors exploit gating mode to improve performances of the quantum channel. As already described a detector working in gating mode is active just for a brief amount of time and it is turned off when it is not expected to receive quantum signals. These devices need to calibrate activation time, that slightly differs due to the unique electrical characteristics in each device. Calibration consists of sending multiple calibration signals to estimate channel length and delay between arrival timing of pulses at different detectors. Activation timing of each detector is scanned independently to find timing when count rate is maximum [63].

Calibration signals contain no information, hence there is no way to check whether signals are sent by the sender or by a third party untrusted source, an eavesdropper. Usually a single pulse per system cycle is used as calibration signal. Here is where man in the middle attack takes place: an eavesdropper may replace the signal with another one containing more than a pulse per cycle with the aim of introduce disparities between the activation timing of different detectors. Ad example, an eavesdropper may send two pulses in the same cycle, one at time  $t_1$  and the other one at time  $t_2$ . Maximum count rate in this case can either be at time  $t_1$  or  $t_2$  randomly, hence activation time of different devices can be quite different. With different activation timings eavesdropper can use other kind of attacks to obtain information about the key, ad example the already explained time shift attack.

To prevent this kind of attack, relationship of activation timings of various detectors must be strictly monitored during calibration process. This monitoring can be performed from software side, removing needs for special hardware requirements.

Another way to avoid this attack is by using measurement device independent QKD [64], which remove all detector side channels, this solution however requires more hardware device and can be more expensive than a classical configuration of detectors.

## 5.6 Other attacks

Exploiting different configurations of practical QKD systems many other attacks may take place.

*Trojan horse* attack takes advantage of the reflected light to gain information about the system [65]. An eavesdropper can inject light pulses into the system and check the different reflections obtained by specific quantum states passing through the system.

To avoid this attack a correct design of the system becomes crucial. Filters, isolators and privacy amplification methods can be employed to ensure information leakage gained to eavesdropper is below a given threshold.

A similar attack is *back-flash* attack. This is a passive attack since eavesdropper gains information about the detector that clicked by looking at secondary photons emitted by a gated InGaAs detector during the avalanche of charge carriers due to a detection event. Also in this case, designing the system with filters and isolators prevent information leakage [66].

*Laser damage* attack consists of damage detector of receiver by injecting a laser light with a power beyond detector limit [67]. This will cause detector to permanently change its characteristics. Depending on how much these characteristics change, eavesdropper may remotely control efficiency and dark count rate of detector. Dark count rate is usually subtracted by the final QBER count. If eavesdropper can control this parameter, he can modify the QBER value and leave his presence unrevealed.

This attack, such as most of the ones that rely on flaws of detectors, can be avoided by employing device independent QKD scheme, in which detectors do not need to be trusted and can even be controlled by the eavesdropper.

*Denial of service* attack is probably the simplest and easily achievable attack. With this attack an eavesdropper can prevent sender and receiver from generating a key. Since quantum channel is always a dedicated line between sender and receiver, this can be easily achieved by cutting the optical fiber composing the quantum channel [68].

Solution to this problem led to development of quantum networks, to route quantum signal over different paths just like it happens with routing in current IP networks.

*Double-click* attack exploit the double-click event. A double-click event occurs when two different detectors are used to detect '0' and '1' values and, for an error, both of them click at the same time. When this happens, if one of the two bit is selected the QBER is increased of 50%. Hence the two parties can agree to discard qubit measured in a double-click event in order to minimize QBER. This opens a window of attack for an eavesdropper who may floods polarization beam splitter of receiver with multiple photons in order to cause a double-click event when receiver makes a measurement using a basis different from the one used by eavesdropper, otherwise he does not disturb measurement. Since all photons measured in different basis from the one chosen by eavesdropper are discarded, at the end sender and receiver will share the same key with eavesdropper.

To solve this problem a simple solution is to not discard qubit measured during double-click event even if it increases QBER [69].

The attacks shown so far can be avoided by taking the right implementation choices. However other studies outlined how, even if an implementation is perfect, it could be possible for an eavesdropper to create vulnerabilities by damaging employed components using a laser [70]. A more general solution to most of the side-channel attack is employment of Device Independent QKD protocols which guarantees a secure communication between two isolated nodes irrespective of the actual implementation of the protocol, even if devices employed are under full control of eavesdropper [46].

In practice DI-QKD solutions are not widely adopted mainly because of the poor key rate they can provide. A more practical solution is represented by *Measurement Device Independent QKD* (MDI-QKD) that guarantees higher key rate under the assumption that the source is trusted [71].

# Chapter 6

## Qiskit framework

### 6.1 Introduction

After having seen how quantum key distribution is realized in terms of protocols and physical devices, it can be worth to look at how programming in quantum computing can be achieved. There are different frameworks that allow quantum circuits programming, each one with different features. However, since the presented work is based on Qiskit, it is the only one that will be discussed here.

Qiskit is an open source SDK for quantum computing. It allows to work with quantum computers at the level of pulses, circuits and algorithms [72]. The work presented here is based on Qiskit mainly because it is the official framework supported by IBM, who allows programming of real quantum computers exploiting the *IBM Quantum Experience* program <sup>1</sup>. Moreover, Qiskit offers a tool to simulate quantum circuit on a local device. Hence any quantum program can be studied and simulated locally and then be sent to a real quantum computer for running. This simplifies a lot quantum programs development since current quantum technology is pretty expensive and cannot be accessed by everyone.

The goal of Qiskit is to accelerate development of quantum applications. In order to do that, this framework has been developed using a simple traditional programming language: python. The framework itself comes as any other python modules and can be therefore installed through *pip* package registry.

Qiskit comes with a comprehensive handbook <sup>2</sup> that explains step-by-step principles of quantum computing in an interactive way, by running proposed circuits. Moreover, it can count on a spread worldwide community that develops examples and offers support. Hence, a rich set of examples and applications are already available online and can be used to explore quantum computing potentialities.

#### 6.1.1 Components

Qiskit framework is organized in four different components:

- *Terra*: Composing quantum programs at the level of circuits and pulses with the code foundation.
- *Aer*: Accelerating development via simulators and noise models.
- *Ignis*: Addressing noise and errors.

---

<sup>1</sup><https://quantum-computing.ibm.com/>

<sup>2</sup><https://qiskit.org/textbook/what-is-quantum.html>

- *Aqua*: Building algorithms and applications.

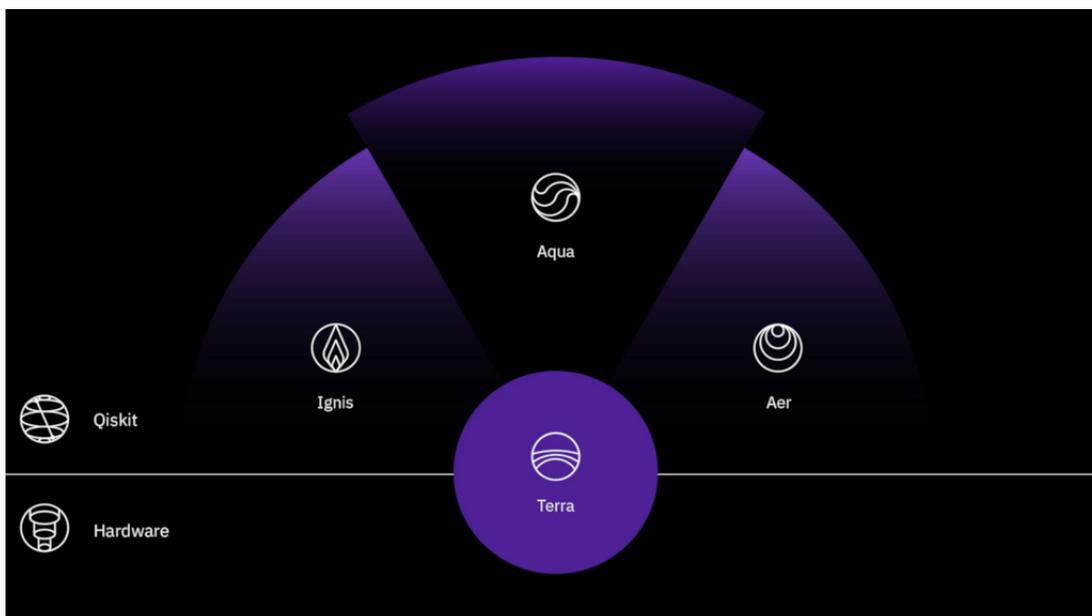


Figure 6.1. Qiskit framework (source: [qiskit](https://qiskit.org)).

## Terra

Terra is the main component on which all other elements lie. It allows to build quantum circuits using gates and pulses. Besides, it is the component to use in order to manage execution of batches of experiments on remote-access devices (e.g. devices accessed through the IBM Quantum Experience program).

Terra is also organized in different modules:

- *qiskit.circuit*. It is the module used to build and run circuits. It allows to select a sequence of quantum operations (gates) on a register of qubits. This register is initialized with all qubits set to  $|0\rangle$  state. Passing through gates, the qubits state evolves in something that cannot be efficiently represented on a classical computer. To extract state information a measurement is performed which maps the outcomes to classical registers. This module collects all the objects and functions to efficiently realize a quantum circuit.
- *qiskit.pulse*. This is the module that allows to work with pulses. Pulses can be sent to a channel (experimental input line) in order to realize experiments that study methods to reduce errors (e.g. dynamical decoupling, error mitigation, and optimal pulse shapes). This is a lower level than circuits and requires each gate in the circuit to be represented as a set of pulses.
- *qiskit.transpiler*. It is an optimization module that, given an input circuit, returns another circuit optimized for running on real devices. On real devices experimental errors and decoherence can introduce computational errors. This module is aimed to reduce the number of gates and the overall running time of a given circuit (while maintaining the desired algorithm) in order to obtain a robust implementation.
- *qiskit.providers*. This is the module aimed to manage experiments on remote devices. It is further composed of four parts:
  - *BaseProvider*. It is a class representing a provider that provides access to a group of different backends. It allows to find a currently available backend such as retrieve an instance of a particular backend.

- *BaseBackend*. It is a class representing a backend that is either a simulator or a real quantum computer. Thus, this is the component in charge of running quantum circuits and returning results. Circuits run by means of jobs that are the input of `BaseBackend.run` method. Jobs will run asynchronously thus `BaseBackend.run` method will return a `BaseJob` object to access the results.
  - *BaseJob*. It is the reference for a submitted job. It allows to retrieve the job status (if it is queued, running or finished), get back job result and more in general control the job.
  - *Result*. This represents the results of the executed job and can be obtained from the remote backend using `result = job.result()`. Result object holds the quantum data. It also allows to deeper analyse raw counts of circuit by using `result.get_counts()` with the target circuit as parameter.
- *qiskit.quantum\_info*. It is a utility module to perform advanced analysis on circuits run on quantum computers (e.g. it allows to analyse data coming from `result.get_counts()` function). It allows to both estimate metrics and generate quantum states, operations and channels.
  - *qiskit.visualization*. This module allows visualization of generated circuits as well as plotting quantum states and histogram of circuit results. It is a utility module to have a quick overview of the designed circuit and its state. Visualization of circuits schemes allows to avoid basic misconfiguration and can be seen as a visual debug tool.

## Aer

Aer is the component designed to simulate quantum computers execution on classical processors. It provides high performance simulator framework that can be accessed through three different simulator backends, allowing execution of circuits compiled in Terra. Besides, it also provides tools for constructing noise models, allowing to perform realistic noisy simulations of errors that occurs employing real devices.

The available simulator backends are:

- *QasmSimulator*. It allows ideal and noisy multi-shot execution of circuits and returns counts on memory. To better simulate different circuits the following methods can be chosen:
  - *statevector*: uses a dense statevector simulation.
  - *stabilizer*: uses a Clifford stabilizer state simulator only valid for Clifford circuits [73] and noisy model.
  - *extended\_stabilizer*: uses an approximate simulator in which circuits are decomposed into stabilizer state terms.
  - *matrix\_product\_state*: uses a *Matrix Product State* (MPS) simulator [74].
- *StatevectorSimulator*. It allows ideal single-shot execution of circuits and returns the final statevector of the simulator.
- *UnitarySimulator*. In this backend, circuit cannot contain measure or reset operations. It only allows ideal single-shot execution of a circuit and returns the final unitary matrix of the executed circuit.

## Ignis

Ignis is a utility component to characterize errors, improve gates and computing in the presence of noise. It provides code to generate circuits for specific experiments giving a minimal set of input parameters. It is based on the following modules:

- *Circuits*. This module provides code to generate a list of circuits for a particular experiment with a minimal set of user parameters.

- *Fitters*. This module collects results of an experiment in order to analyse and fit them according to the physics model describing the experiment.
- *Filters*. This module is returned by fitters only in particular experiments. It can be used to mitigate errors in other experiments using calibration results it contains.

Three types of experiments can be performed using this component:

- *qiskit.ignis.characterization*: characterization experiments targeted to measure parameters in the system (e.g. noise parameters).
- *qiskit.ignis.verification*: verification experiments aimed to verify gates and small circuits performances. These experiments provide the information to determine performance metrics such as the gate fidelity.
- *qiskit.ignis.mitigation*: mitigation experiments are designed to generate mitigation routines by running calibration measurements. The results of these measurements will be processed by a Fitter and will output a Filter than can be used to apply mitigation to other results.

## Aqua

Aqua is the highest level component that allows to build algorithms for quantum computers. On top of these algorithms, application for chemistry, optimization, finance and artificial intelligence have been already built in order to exploit quantum computers as accelerators for specific computational tasks <sup>3</sup>.

### 6.1.2 How to simulate a quantum circuit

In order to simulate a quantum circuit it is needed to design it at first. This can be done by using qiskit Terra component.

A quantum circuit can be composed of both quantum and classical registers. Usually, classical registers collect output of the circuit whilst quantum register is where the quantum operations are performed. Circuits are represented by module *qiskit.circuit* and, in order to retrieve and model an instance of this module the following instruction needs to be called:

```
circuit = QuantumCircuit(3, 1)
```

Where the input parameters represent the number of quantum (first parameter) and classical (second parameter) bits the circuit is composed of.

After the circuit is created it is possible to add gates in order to perform specific operations. Gates can be added by calling related method on the `circuit` object returned after circuit initialization. For example, considering the Hadamard gate, this can be added to the circuit by calling method `h` on circuit as follows:

```
circuit.h(0)
```

Input parameter in this case represents the qubit affected by Hadamard gate.

Besides Hadamard gate, `circuit` object allows to add many other gates by using related methods. The full list of methods representing available gates can be found on qiskit documentation <sup>4</sup>. Here is reported a list of the most basic ones:

- `x(qubit)`: applies the Pauli-X gate to the specified qubit of the circuit.

---

<sup>3</sup><https://github.com/Qiskit/qiskit-aqua>

<sup>4</sup><https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html>

- `y(qubit)`: applies the Pauli-Y gate to the specified qubit of the circuit.
- `z(qubit)`: applies the Pauli-Z gate to the specified qubit of the circuit.
- `s(qubit)`: applies the S gate to the specified qubit of the circuit.
- `t(qubit)`: applies the  $\frac{\pi}{8}$  gate (T) to the specified qubit of the circuit.
- `cx(control_qubit, target_qubit)`: applies the CNOT gate to the specified qubits of the circuit.
- `cz(control_qubit, target_qubit)`: applies the CZ gate to the specified qubits of the circuit.
- `toffoli(control_qubit1, control_qubit2, target_qubit)`: applies the Toffoli gate (CCNOT) to the specified qubits of the circuit.
- `measure(qubit, cbit)`: measures quantum bit into classical bit.

Once desired gates have been added, it is possible to have a visual preview of the resulting circuit by using `draw` method, that will show a graphical representation in order to check interactions between qubits and gates in the circuit. If resulting circuit is correctly designed it is possible to proceed with simulation phase.

Simulation is performed exploiting qiskit Aer component. The first step in this case is to select the backend that allows to run the circuit. After circuit execution statevector backend returns the quantum state, which is a complex vector of dimensions  $2^n$ , where  $n$  is the number of qubits of the circuit. Unitary backend returns the  $2^n \times 2^n$  matrix representing the gates in the circuit. It works only if all elements in the circuit are unitary operations. However, if measurement operation needs to be performed on quantum bits (that is the usual condition for most of quantum circuits), qasm backend must be employed. The selected backend can be retrieved from Aer component as follows:

```
Aer.get_backend('qasm_simulator')
```

To actually perform the simulation, quantum circuit and backend must be passed as parameters to `execute` function. Within this function it is also possible to specify the number of times circuit must be executed in order to build up statistics about the distribution of the bitstrings. The output of `execute` function is a *BaseJob* object that can be used to retrieve execution results in a *Result* object. From this latter object the function `get_counts` can be called to retrieve the aggregated binary outcomes of the submitted circuit.

All the steps here described will be shown in a practical example in the next section.

### 6.1.3 First quantum circuit example

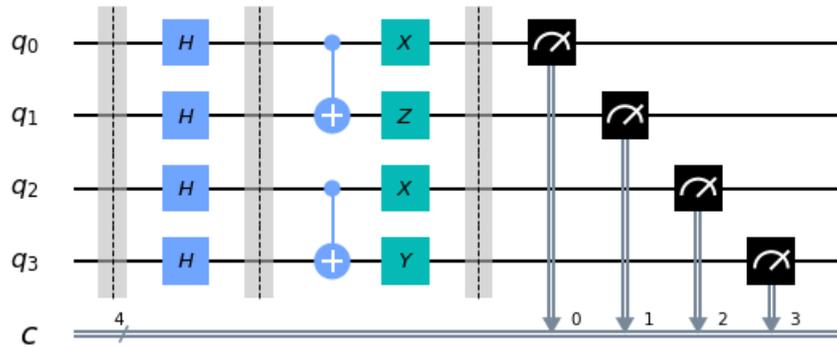
In order to show a complete example of circuit simulation, the example circuit in figure 2.4 will be executed. To summarize, the steps needed to simulate this quantum circuit are:

1. Create a quantum circuit with desired quantum and classical bits. In this case 4 quantum bits are needed, which will be measured into 4 classical bits:

```
circuit = QuantumCircuit(4, 4)
```

2. Prepare inputs by creating superposition with Hadamard gates. To replicate desired circuit an Hadamard gate will be applied to each one of the quantum bits:

```
circuit.h(0)
circuit.h(1)
circuit.h(2)
circuit.h(3)
```

Figure 6.2. Output of function `circuit.draw('mpl')`.

3. Add gates required by quantum algorithm. In this case it is needed to add a CNOT gate between qubit 0 and 1, a CNOT gate between qubit 2 and 3, an X gate for qubits 0 and 2, a Z gate on qubit 1 and an Y gate on last qubit:

```
circuit.cx(0, 1)
circuit.cx(2, 3)
circuit.x(0)
circuit.z(1)
circuit.x(2)
circuit.y(3)
```

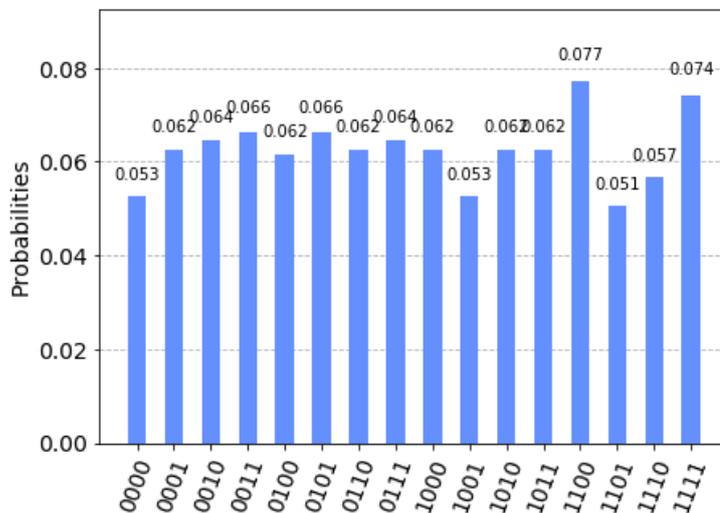
4. Add measurement gates to map quantum bits on classical ones:

```
circuit.measure(range(4), range(4))
```

5. When circuit is ready, it can optionally be drawn for a visual check:

```
circuit.draw('mpl')
```

Using 'mpl' as argument will invoke *matplotlib* library for a nicer visualization. However, this function can also be called without arguments. The resulting drawn can be seen in figure 6.2.

Figure 6.3. Output of function `plot_histogram(counts)`.

6. Retrieve the desired simulator backend:

```
backend = Aer.get_backend('qasm_simulator')
```

7. Execute the circuit and get results back. Circuit will run 1024 times that is the default number of times to get a reliable statistical distribution of the bitstring.

```
job = execute(circuit, backend, shots=1024)  
result = job.result()
```

8. Get the aggregated binary outcomes of the measurements:

```
counts = result.get_counts(circuit)
```

9. Optionally, instead of printing and analysing raw data counts, it is possible to plot them with the following function:

```
plot_histogram(counts)
```

The result is shown in figure [6.3](#).

# Chapter 7

## QKD in softwarised networks

### 7.1 Introduction

To effectively exploit the advantages of a QKD system, it can be useful using it inside softwarised networks. In such a way multiple applications can access the system and share different keys between them without actually possess a QKD device. The aim of this chapter is to give an overview of possible applications for a QKD system in a softwarised network.

Before describing possible use cases of QKD systems, it may be worthy to define the meaning of softwarised networks. Softwarised networks refer to networks in which services do not rely on particular hardware solutions. Resources in softwarised networks are virtualised in order to be accessed from different points within the network without the need of implementing specific producers protocols to access them. In this way different kind of resources can be abstracted and used through the same interface, increasing flexibility in service supply.

Among the different softwarised network paradigms it is possible to find:

- *Cloud computing*: this paradigm is aimed to virtualise computation [75].
- *Software Defined Networking* (SDN): this paradigm consists of networks virtualization [76].
- *Network Functions Virtualisation* (NFV): this paradigm is aimed to virtualise network functions [77].

A QKD system can be integrated within a NFV environment. In a NFV architecture (figure 7.1) network functions are virtualised in different building blocks that may be chained to create communication services. Thus, NFV allows flexible provisioning, deployment, and centralized management of virtual network functions. The NFV architecture is composed of: a *NFV Infrastructure* (NFVI) that manages different physical resources and virtualises them to let them be available for network functions execution; *NFV Management and Orchestrator* (MANO) that collects and manages all virtualisation-specific management tasks needed by the NFV framework; the *Virtualised Network Function* (VNF) that represents the software implementation of a network function.

Virtualised network functions can be firewalls or any other security application that may require keys. In such a scenario VNF represents the Secure Application Entity (SAE) defined in chapter 4 and are the entity that may require keys to a QKD system. In order to provide quantum distributed keys, the underlying level must be able to manage a QKD system by means of the virtualisation layer, that should abstract the access to the physical hardware resource representing the QKD devices. An example can be represented by IPSec protocol, in this protocol two instances need to rely on keys in order to ensure a secure communication. The two IPSec instances can be seen as two SAE and thus they can represent two VNFs. This instances can require a quantum key exchange to the virtualisation layer and obtain a shared key that can then be used as preshared key in the IPSec protocol.

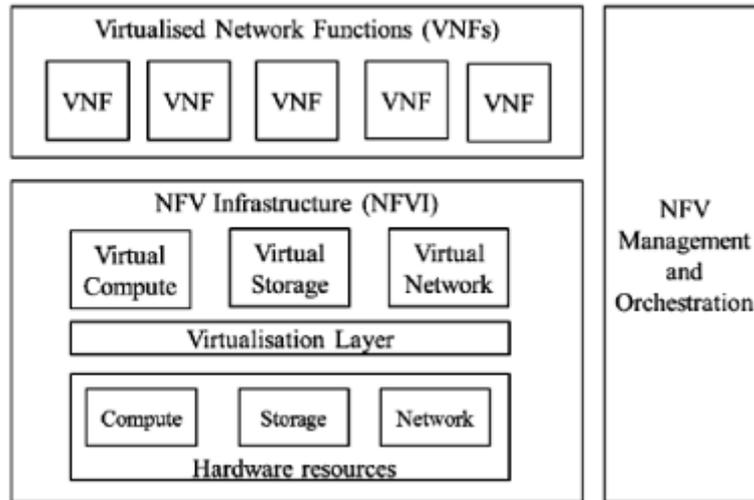


Figure 7.1. NFV architecture (source: ETSI).

In order to be able to insert QKD in a NFV, it is important to be able to abstract the hardware implementation from the logical one. In order to achieve such results it is important to be able to define protocols and architectures that can be used in such distributed infrastructures. In such scenario the role of simulator becomes crucial. Simulator can be used to design possible integrations by using different protocols, outlining the main issues and challenges of such integration. Besides, simulator allows to study specific scenarios and related attacks in order to design targeted solutions with real devices.

NFV MANO includes a component named Virtualised Infrastructure Manager (VIM). This component manages NFVI resources (Compute, Storage and Network) at high level. Hence, in order to integrate QKD in a NFV environment, VIM component should be patched to use this technology. This would allow a virtual compute to require a quantum distributed key to the system and use it to communicate with another instance in another NFV architecture. Besides, since NFV architecture extends cloud architecture, an integration on this kind of systems would imply a simple integration in generic cloud architectures as well.

The scope of this work is to design an ETSI standard compliant key server capable of exchanging quantum keys and distribute them in a softwarised network. After having seen how softwarised networks are composed, it becomes clear the need to realise a QKD Key Server as much flexible as possible in order to be able to be integrated in different architectures without relying on particular hardware resources. The proposed work was not actually integrated in a real cloud environment, but it has been designed in order to easily achieve such result.

## 7.2 Use cases

A set of possible use cases are described in ETSI GS QKD 002 document [52]. These example regards both two instances key exchange as well as key exchange in a network.

### 7.2.1 Two instances key exchange

The simplest example regards an offsite backup for business continuity. In this case an enterprise owns a private network in which operates a primary site for main data processing and a secondary site to regularly perform backup of the primary site. The goal is to use the secondary site to recover information in case of data loss on the primary site. Since data processed are confidential, the enterprise should employ an encryption system to securely transfer information, hence a QKD link encryptor may be used. Cryptographic keys should be exchanged between primary and secondary

site with a QKD link and fed into a link encryptor which encrypt traffic by using a symmetrical block cipher. In this use case all QKD devices are owned by the enterprise and are located inside

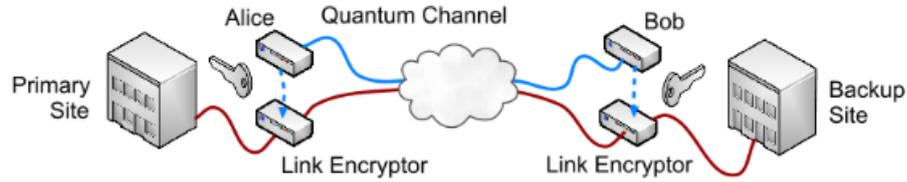


Figure 7.2. Offsite backup use case (source: ETSI GS QKD 002 [52]).

its security perimeter. In this way the enterprise may autonomously create and manages keys. Even if this can seem an overhead, the enterprise may benefit of different advantages such as the secure data transfer, the transparent encryption (application level does not need to manage keys) and no further processing time at the server of the enterprise.

Another interesting use case for exchanging keys between two instances is the one in which the two sites are separated by a long-haul distance. In this case the two sites must guarantee very high security requirements since they cannot use intermediary nodes like in the case of a QKD network (as can be the case of governmental sector). A solution is to use a satellite that passes over the two sites (A and B) once daily, enabling them to share a common secret. The secret can then be used in a symmetrical encryption scheme to secure data transmission between them. When a satellite is employed, communication runs via free space links, which are able to exchange keys over long distances and high key rate distribution rate. In this scenario when the satellite passes over ground station A, it exchanges a symmetrical secret  $a$ . Later, when it passes over ground station B, it exchanges another symmetrical secret  $b$  with this station. Subsequently, it encrypts secret  $a$  and  $b$  with one time pad scheme (basically it performs  $a \text{ XOR } b$ ) and sends this ciphertext back to site B through a classical communication channel. Site B can then easily recover secret  $a$  which A already possess. Thus site A and B share the same secret. In such a scenario it is required that satellite is a trustworthy QKD system since

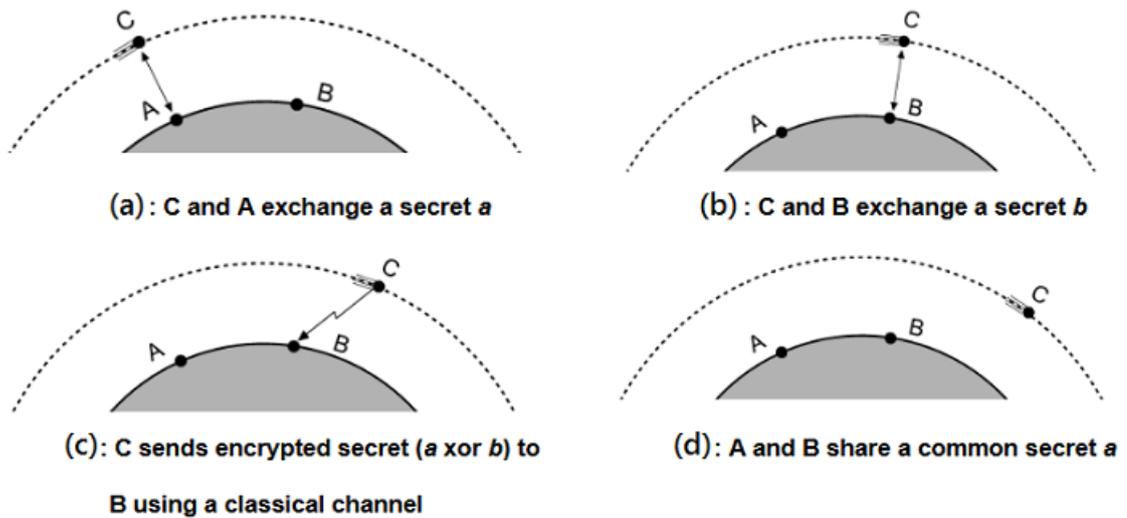


Figure 7.3. Long-haul use case (source: ETSI GS QKD 002 [52]).

it will also share the same secret belonging to A and B sites. Even if this configuration guarantees a high level of security, it offers a tiny window of opportunity for QKD since the satellite is a fast moving target. This can also be viewed as an advantage considering the physical integrity of device, but becomes a disadvantage if a malfunction or failure requires repair. Ideally this scenario should provide an additional redundant key distribution channel as a fall back solution.

Different studies proposed a variant of this use case which employs the use of drones rather than satellites. These studies tried to define low weight and low size QKD system that may be employed in airborne platforms such as drones or aircraft [78].

QKD may also be employed for security services between nodes of a backbone network. Metropolitan area networks are logically divided into backbone and access parts. Backbone is specialized in high speed communication and it is connected to access part through a point to multipoint network, which has one end connected to the backbone while the other gives service to several clients. Current networks are evolving towards optical fiber. The idea here is that any fiber link in a network provider infrastructure has the potential resource of a quantum channel. Current metropolitan area networks widely use *wavelength-division multiplexing* (WDM) to multiplex and route different channels into a single fiber. In such a scenario one specific channel can be used as quantum channel of a QKD system. This channel can produce a stream of symmetrical secrets on ends of the backbone link. These secret keys can be used for cryptographic tasks on the level of the infrastructure provider. Also in this case, just like the first one, the infrastructure provider owns the backbone network, implying that backbone nodes are trusted.

Exploiting the same principle, already existing fiber links can be used as QKD channel even for realizing high security access network. In Fiber to the Home access network architectures a *Passive Optical Network* (PON) connects one *Optical Line Terminal* (OLT) with multiple *Optical Network Units* (ONUs). The OLT is installed at the service provider facility whilst ONUs are installed near end users. Figure 7.4 shows an example of such architecture. Usually, this configuration

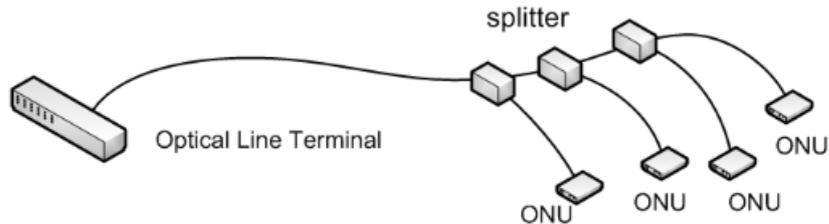


Figure 7.4. Passive Optical Network architecture (source: ETSI GS QKD 002 [52]).

implies that each ONU sees the entire downstream from the OLT, even the content that is not intended for it. This is usually fixed encrypting communication with symmetrical keys which may be distributed through smart cards or with asymmetrical methods (e.g. Diffie-Hellman). Instead, the same path used to transfer classical information can be used to exchange quantum information encoded in single photons or weak pulses. A single photon can only arrive to one ONU, ensuring the same secret is exchange only between two parties. In such scenario final users must trust service provider as regards the security of the downstream content. However, the relatively short distances of this architecture ensure a high QKD key rate.

The same architecture can be used for the authentication of sensors and actuators and their data in a local sensor network as shown by figure 7.5 In this case a central control station operates on one side of a common QKD link while sensors and actuators act as instances of the other side. Like for the above case the central station in this architecture is capable of sharing keys with any of the peripheral QKD devices.

### 7.2.2 Multi instances key exchange

Interesting use cases regards the ones in which already existing solutions integrate QKD protocols. An ideal case would be the one in which the QKD system is integrated in one of the layer of OSI model in order to integrate safe communications in a network. In such case, data pass through the network in a reliable infrastructure that allows to create areas in which requirements of integrity, confidentiality and authenticity are satisfied. This kind of network is named *QKD network* (figure 7.6).

It would be possible to integrate QKD at various level of the OSI model depending on the availability of the QKD system and the information that needs to be secured.

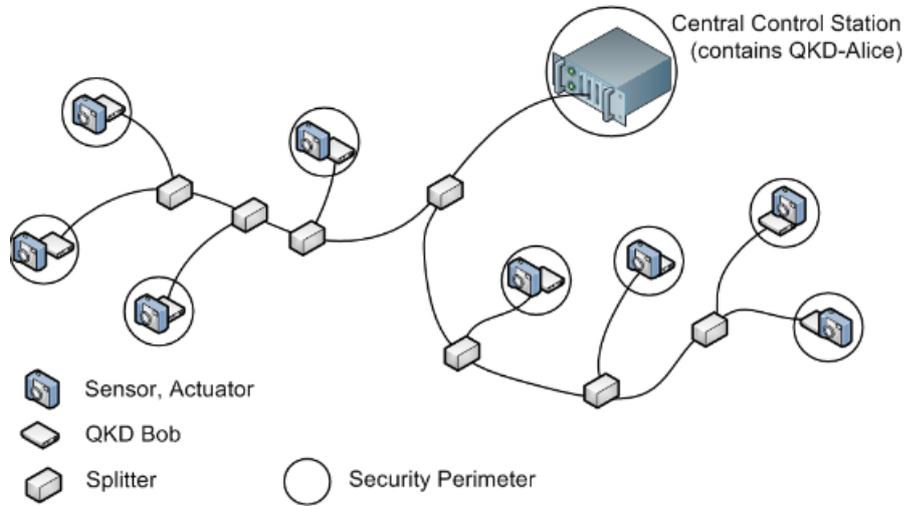


Figure 7.5. QKD authenticated sensor network (source: ETSI GS QKD 002 [52]).

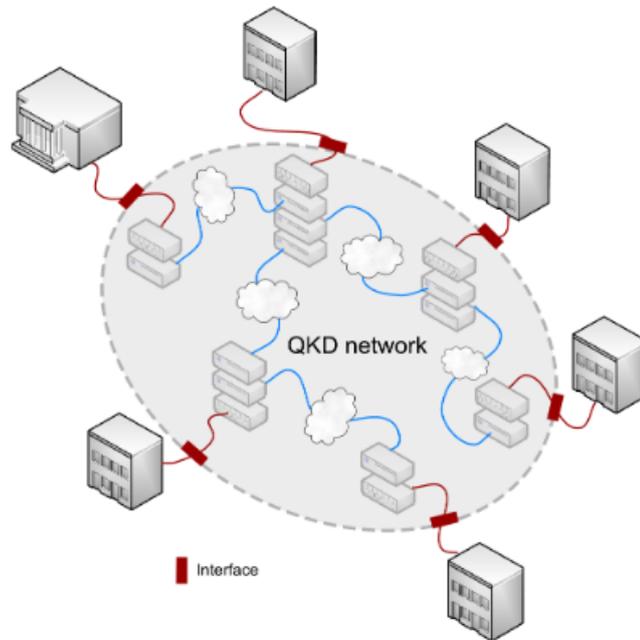


Figure 7.6. QKD Network (source: ETSI GS QKD 002 [52]).

Integration can be performed on *Data Link Layer*. In this case QKD can be used as part of *Point to Point Protocol* (PPP) or *MACsec* protocols, which are two widely used protocols for local area network communication. However, at this level of OSI model most of the communications are point to point, hence it is possible to refer to some of the use cases presented in previous paragraph to have an idea of how such integration can be used.

A different scenario can be outlined if QKD is integrated in *Network Layer*. Here, IP packets are authenticated and encrypted by using the protocol *IPsec*. Specifically IPsec suit uses another protocol named *Internet Key Exchange* (IKE) to set up security associations. This protocol uses Diffie-Hellman public key exchange or, alternatively, pre-shared keys to mutually authenticate communication parties. Diffie-Hellman can be replaced by a QKD system in a modified IKE protocol in order to provide shared secret for IPsec payload.

In level four of OSI model, security is usually established by using *Transport Layer Security*

(TLS) protocol. A variant of this protocol is represented by *TLS Pre-Shared Key* (TLS-PSK) in which keys shared in advance between the two parties are used to establish the TLS connection. In such scenario, keys exchanged through a QKD link may be employed in order to secure communication for the high level applications.

QKD may also be integrated in *Application Layer* of the OSI model in all those applications that may require secure communications.

As generic example consider a company may want to connect one or more data centers with branch offices. Current solutions involve use of *Virtual Private Networks* (VPNs) based on IP sec or TLS to authenticate and encrypt traffic between data centers and branch offices. In this scenario the single network connections between sites can be secured with QKD link encryptor. The generated keys can be used to encrypt and authenticate data on the OSI data link layer. In such scenario it is also possible to employ a centralised key server in order to share the keys that will be fed in the link encryptors as shown in figure 7.7.

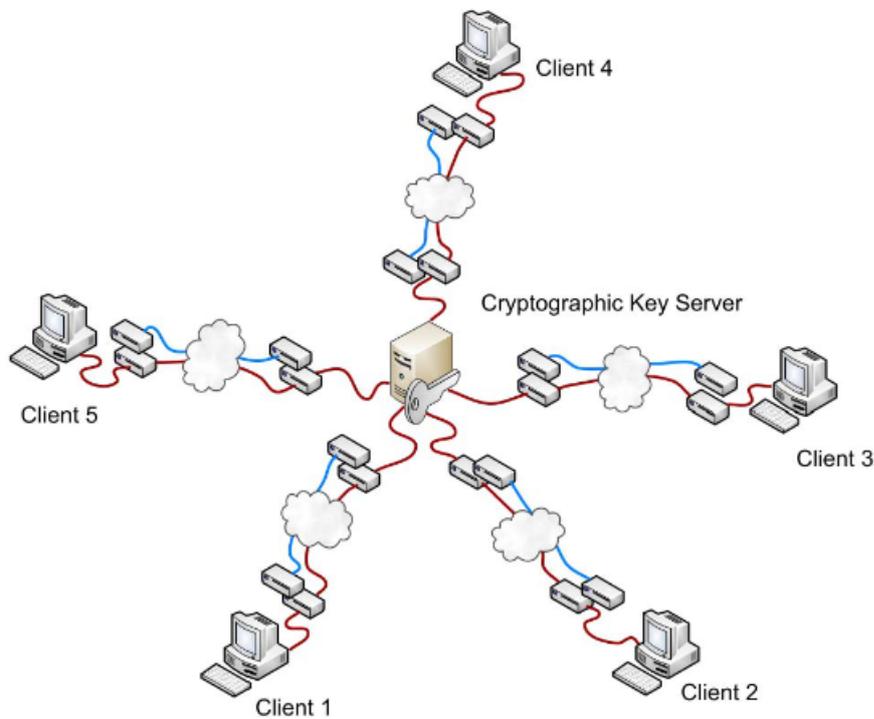


Figure 7.7. Central key server architecture (source: ETSI GS QKD 002 [52]).

Integrating QKD in OSI model is a way to move from classical to quantum cryptography seamlessly, without the need of change already existing architectures and protocols. Besides, the possibility to integrate QKD at various level of the model allows to gradually move to quantum cryptography by integrating QKD systems step-by-step where needed.

## Chapter 8

# QKD simulator

QKD simulator is the basic building block of this project. It allows to perform a key exchange among different parties without employing real devices. Real devices are usually expensive and technology to realise QKD is not yet easily accessible. A simulator allows to study new protocols or test existing ones even without a physical access to related devices. Moreover, simulator can be extremely useful to map the QKD physical phenomena into mathematical models. With these models it is possible to perform consistency tests of QKD systems without having a testbed. All QKD components can be mapped to mathematical models: single photon sources, decoders, quantum channels and also qubits measure and analysis.

Already available examples of QKD simulator focus their attention on the protocols they try to simulate. In particular it is possible to find examples of BB84 and E91 protocols that share keys between two software instances on the same machine. This is done mainly to show the steps involved in these protocols. Starting from these implementations a new kind of simulator has been here developed. This simulator is capable of sharing keys between two different machines, by simulating a quantum channel over the internet. Every component in this implementation is split by the others so that it is possible to introduce specific mathematical models for a component without affecting the other ones.

### 8.1 Simulator Design and Architecture

As said, simulator has been designed in different building blocks, each one representing a different component involved in a QKD process. These building blocks are depicted in figure 8.1 and consists of the following components:

- *Source*. It represents one of the two parts involved in the key exchange. It performs different steps depending on the protocol used and usually is the one who starts the communication. This block can be used to model qubits sources.
- *Detector*. It represents the destination of the key exchange. Even steps here performed depend on the employed protocol. Inside this block different kinds of decoders can be model.
- *Channels*. Quantum and classical channels forward both quantum and classical messages from source to destination and are under full control of eavesdropper (Eve). These channels have been thought as separate blocks in order to allow different kind of attacks to be performed, such as introduce different kind of noise modelling.
- *Singlet source*. This block is used when entanglement based protocols must be used during the exchange. This block is in charge of generates entangled qubits pair and distribute them to requesting entity. In this scenario qubits are directly requested by the channel in order to allow simulation of specific attacks on this source.

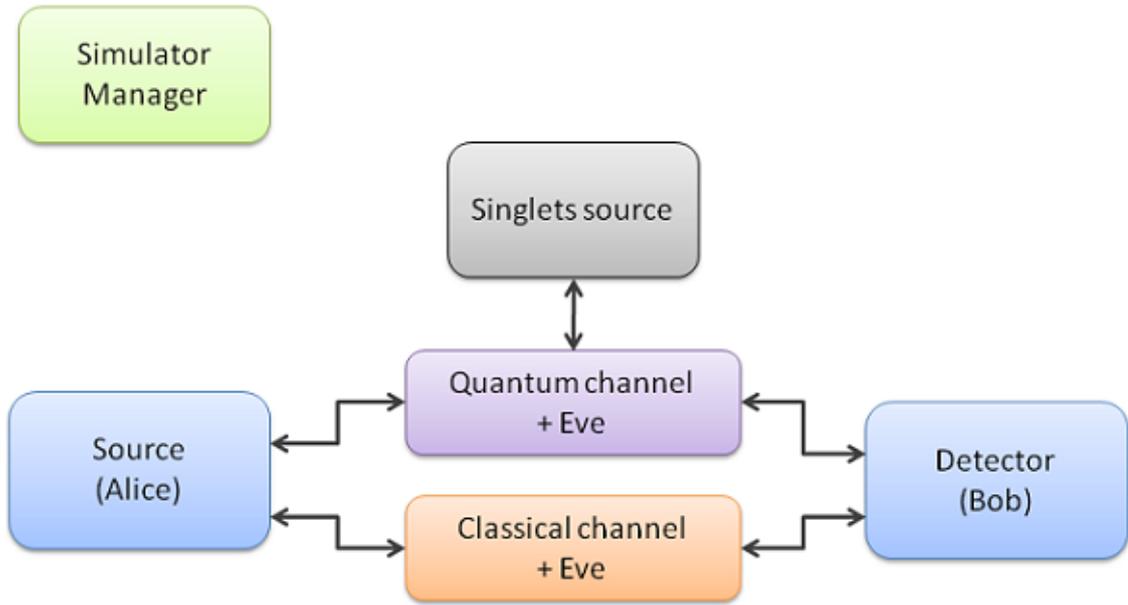


Figure 8.1. General simulator architecture.

- *Simulator manager.* The manager is an additional module that allows to send commands to the above systems. It is possible to request key exchange with specific protocols and modify channel settings in order to enable/disable attacks. The manager was designed to provide an user friendly interface to simulator environment and can be used to collect data about simulator usage.

In order to connect source and destination on different machines, both quantum and classical channel are realised through a web server. This because web servers are easily reachable from other applications and there are many ready-to-use solution which allow to realise them. Specifically, in this implementation *Flask* module has been used, which allows to build and run a server with very few configurations. Since the classical channel does not need to data confidentiality, web server does not have the need for particular settings. The only thing that must be considered is that public classical channel must be authenticated. Different kind of authentication methods have been considered. At first a certificated based authentication with PKI has been evaluated. However, this solution has been discarded mainly because it leans on algorithms (RSA mainly) that have already been proved to be weak against quantum computing. Hence post quantum cryptography has been studied.

In this scenario authentication is made on each data exchanged through the web server, by signing sent data with a private key and verifying received data with the public key of the caller. Each one of the parties involved in the communication must possess its private key. Public key can instead be released publicly. For simplicity, each instance possess a copy of the public key of other instances, however these data can also be managed by an authentication authority that the instances will query when needed. Among the different post quantum cryptography algorithms, the first solution studied was represented by NTRU. NTRU is composed of two main components: *NTRUEncrypt* and *NTRUSign*, that allow to encrypt data and sign them respectively. In order to achieve authentication, sign operation was analysed. However, while NTRUEncrypt is still a valid solution for asymmetric encryption, NTRUSign has been found to have some flaws and a study demonstrated that it can be easily broken [79]. The choice was then moved to *SPHINCS+*, that is specifically designed as signature scheme and has different python implementations already available. *SPHINCS+* has been successfully integrated and tested inside the system.

Another approach involving symmetric cryptography as been considered in this phase. This approach uses preshared keys in order to perform data authentication. This is realised by using authenticated encryption provided by AES-GCM protocol. This solution consists of sending

encrypted data over classical channel in order to exploit the authentication feature it carries with it. AES-GCM has been proven to be relatively secure against both classical and quantum computing if 256 bits keys are employed. The only drawback of this solution is the needs of a set of keys that will be used to encrypt/decrypt messages over the channel (keys must be changed to ensure security). This problem has been solved by providing the parties involved with a set of preshared keys that can be increased during time with keys directly exchanged by simulator. AES-GCM authentication can be selected in place of *SPHINCS*<sup>+</sup> by changing simulator settings.

Quantum channel can be accessed exploiting classical channel. In this case there is no need of authentication nor encryption for this channel. Basically, since this is a simulated environment, there is not a different endpoint for the quantum channel. In order to access it, another implementation of this simulator will call particular functions in classical web server, which then realise quantum operation by exploiting qiskit framework. All quantum information are forwarded as responses in a classical client server architecture, hence they will actually be exchanged as classical information. Qubits over classical channel are sent by means of *Statevector* objects. Statevectors are qiskit objects that allow to fully represent the state of a qubit by using classical data. The state saved in these objects can evolve if they are let them pass through a quantum circuit in the same way real qubits would do. This allows to let qubits state evolve even if circuits used for polarising and measuring are in two different machines. It is enough to prepare a statevector that will then be polarised by using a given circuit. The output of this process is another statevector that will be sent over the classical channel by serialising the object with a bytes string. The object received from the destination will then de-serialised and measured with selected circuit. This process allows to simulate a real quantum key exchange procedure.

Web server is thus an interface to the internal data manipulation for both classical and quantum data. This manipulation is made by a core component that use *Qiskit* framework in order to perform operations on quantum bits. The internal component hence use quantum circuits as the one presented in chapter 6 to measure and manipulate statevector objects.

Implementation of core component depends on the required protocol. Currently implemented protocols are *BB84* and *E91*. However, new protocols can be added with little modifications to the source code. Currently both source and detector are composed with the same elements: they share the same source code and can act both as source or destination of the key exchange. A detail of components involved inside source and destination can be seen in figure 8.2. Besides the

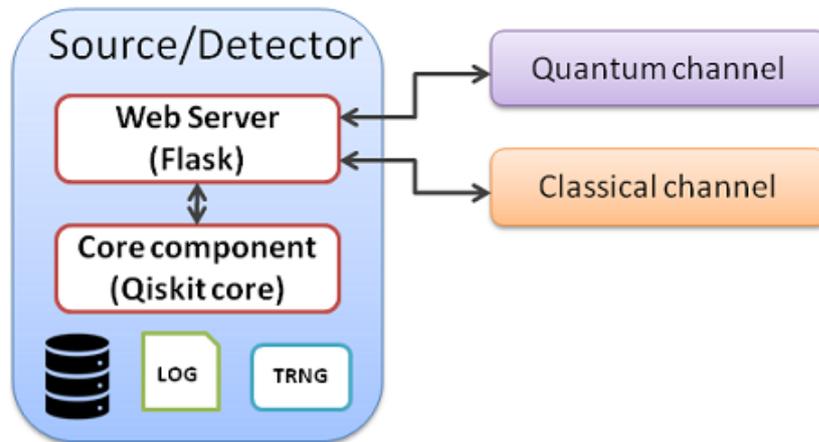


Figure 8.2. Source (and detector) internal components.

already discussed web server interface that allows to access quantum and classical channel, source and destination lean on other external modules to ensure required functionalities:

- *Storage*. Storage is used in more than a scenario mainly to ensure status maintenance among different calls to the web server interface. The idea is that, when source module receives a key exchange request from a higher level module (or the simulator manager), it can autonomously reach the target simulator through its web server and start the key

exchange. When the key exchange is complete, the source that started the exchange can return the key to its high level module. The detector must instead store the exchanged key until some other module will require it. Hence a storage is needed to securely store keys. Besides the exchanged keys, there is the need of a secure storage also to store keys used to authenticate messages over classical channel if AES-GCM encryption is used. Besides, all these keys need to be synchronised between the two modules involved in the exchange. To securely store keys *Vault* service is used. Data related to keys are instead saved into a *mysql* database is used.

- *Log*. Source and detector also contain a log component to save all requests received on web interface, as well as to log any error that may occur. This log is realised through a file that is overwritten every time web server interface is instantiated.
- *True Random Number Generator (TRNG)*. Encoding, measuring and all the operations that lean on a random component are based on a module that simulates a true random number generator. The module is able to generate a random string of bits of desired length. This is achieved by creating a quantum circuit in which the input qubits are set in superposition state by using Hadamard gates on them. Subsequently, a measurement is performed on each qubit, returning a random result. Since this is just a simulation the quantum properties allowing true random number cannot be guaranteed and the generated number must still be considered pseudo random number. However, the modularity of this implementation allows to substitute the simulated true random number generator with an instance of a true device, in order to obtain true random number with a minimum effort when real devices will be employed.

All above components are used when requested from implemented protocol. Specific implementation details of implemented protocols are here described.

## 8.2 Simulated protocols

The proposed solution has been designed in order to ensure maximum flexibility. Hence, simulation of new protocols can be easily added by modify the building blocks this implementation is composed of. Proposed simulator however allows to select between two protocols that have already been implemented: *BB84* and *E91*.

### 8.2.1 BB84

Simulator performs all steps required by BB84 algorithm: it generates qubits and randomly encodes them in two possible basis, sends the qubits to the target of the exchange who will randomly measure them in the same basis, check basis used on the public channel in order to sift the key and calculate QBER to check for eavesdropper.

The possible basis the algorithm allows are  $|0\rangle$ ,  $|1\rangle$  and  $|+\rangle$ ,  $|-\rangle$ . In order to retrieve the correct value from a qubit encoded with these basis the Pauli-X gate should be applied if the qubit is encoded in  $|0\rangle$  or  $|1\rangle$ , otherwise the Pauli-Z gate should be applied if the qubit is encoded in  $|+\rangle$  or  $|-\rangle$ .

key sifting is quite straightforward. Basis table used during measurement (or encoding) can be authenticated using one of the two authentication methods provided (post quantum or preshared key) and it is sent over the classical channel to the counterpart. The qubits correspondent to different values of basis tables are discarded from the final key, all the other ones are kept.

Finally, key is verified. This step involves comparing part of the key on the public channel and check the number of errors that occur. In order to have a reliable result, selected bits should be distributed along the whole key, besides a number of bits representative of the key length should be selected. Since selected bits are sent through the public channel, they will be discarded from the final key. In order to keep track of all these requirements, when the simulator is requested to

exchange a key with a given length, it will add  $\frac{1}{3}$  of the key length to the final number of bits the key is composed of. This  $\frac{1}{3}$  more will be used to check for errors in the exchange. The bits will however be picked randomly along the key.

Usually quantum channel allows communication in one direction only (one end of the communication owns photon sources and encoder whilst the other end own detectors). However, in this implementation QKD device is supposed to have a bidirectional communication system that allows each party to independently send and receive a quantum communication. Thus both Alice and Bob can start a quantum key exchange. This has been done mainly to simplify integration of this system without the need of distinguish between sender and receiver. However if a real QKD device needs to be employed, it will be enough to change its implementation of the function in charge to exchange the key in order to differentiate between sender and receiver. Eventually, device-to-device communication can be modified in order to meet requirements that will emerge from this differentiation. However, this is an internal implementation of the QKD device and will not interfere with the defined interfaces as well as with higher level modules interaction.

### 8.2.2 E91

E91 protocol has been simulated trying to replicate the same steps foreseen by the protocol. In this protocol it does not matter who produces the entangled qubits: they can be sent by one of the two communication parties (Alice and Bob) as well as an untrusted third party or the eavesdropper itself. In this scenario the choice was to let a third party produce the qubits in entanglement state (singlet source block in figure 8.1). Having a separate module allows to model different scenarios by simply changing a setting inside the module (or by changing the whole module if needed). Even if this architecture allows to have different models as singlets source, right now this module is pretty straightforward: it simply produces entangled qubits and provides them to anyone who requests them. In order to reduce execution time, the module produces a predefined number of qubits beforehand in order to be immediately ready when requested. If more than pre-produced qubits are requested, the module produces the exceeding number upon request.

Singlets source module is directly connected to channel. This choice was made to let channel manipulate qubits if desired. As said, E91 protocol allows the qubits to be under control of an adversary before being sent to destinations. In this way it is possible to model attacks inside the channel before sending the qubit pairs to Alice and Bob.

Ideally, the workflow the system should run consists of the following steps:

- A new key exchange request arrives to Alice (or Bob) who request an appropriate number of entangled qubits to the channel.
- Channel requests the number of qubits required to singlets source module who replies with desired data. In this phase, channel can manipulate received qubit if desired.
- Channel forward received qubits to Alice and Bob who then perform their measurements.
- Alice and Bob compare chosen basis during measurement on the classical channel in order to discard qubits measured in different basis.
- Alice and Bob calculates CHSH correlation value to check if entanglement correlation is valid ( $-2 \leq CHSH \leq 2$  if qubits are not entangled).

When designing this simulation, a huge limitation of simulator came up: entanglement cannot be simulated through distributed nodes. In particular, after an entanglement qubit pair is created, if they are serialised and sent over a net, correlation is lost. This is a huge limitation for this particular protocol: measurements will return completely random results and CHSH value will always be found under the security threshold.

A more correct simulation can be performed if measurements for both the parties are performed in the same machine. In this case it is possible to maintain entanglement property if the two measures run in the same circuit as shown from figure 8.3. If two entangled qubits are sent to

different machines, there is no way to know the measurement outcome the other party has got. Hence, in the case of a distributed environment (a real case, since no exchange take place if the two parties can communicate directly), it was not possible to overcome this limitation with a simulator.

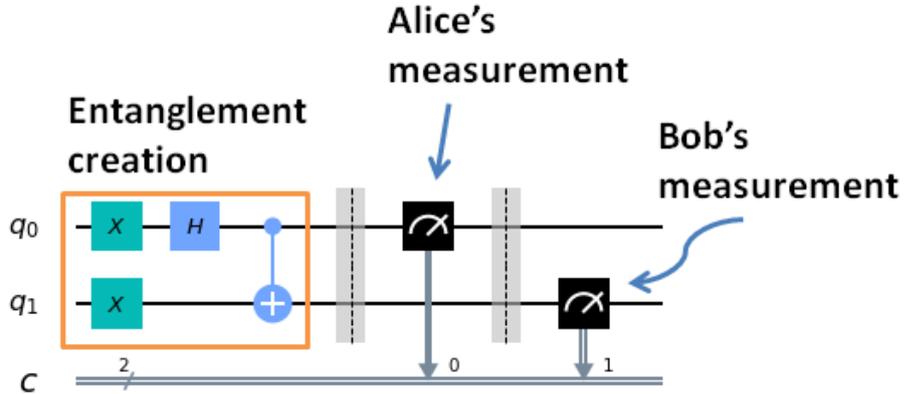


Figure 8.3. Simulated circuit for E91 protocol.

As case study, E91 protocol has been implemented anyway with some differences from the ideal case. In particular, after entanglement qubits creation, channel requests basis table to Alice and Bob and performs measurement on its own. In this way Alice and Bob have control over the basis to be used during measurements, but they do not actually measure qubits. After measurements, channel returns to Alice and Bob the generated key, together with the CHSH value to evaluate entanglement correlation. Even if this implementation differs from E91 protocol involving real devices, it can still be useful as case study to better understand the protocol and models attacks or other scenarios around it.

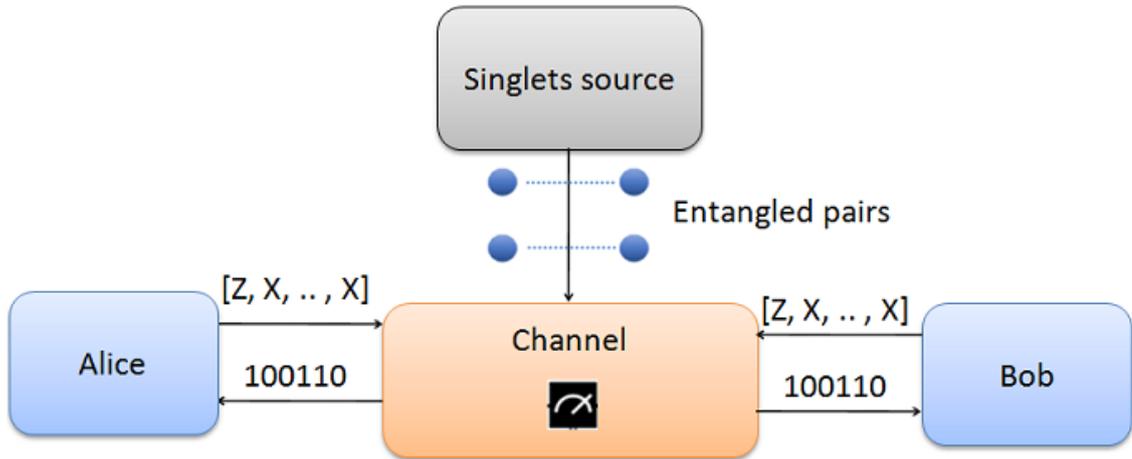


Figure 8.4. Scheme of E91 simulator.

### 8.3 Interfaces

Simulator is basically composed of three main interfaces:

- Web server interface: this interface is used by other instances of the simulator in order to exchange keys. As already said, this interface collects both quantum and classical channels.

- Module interface: this interface is used by high level modules to require key exchanges.
- Manager interface: this interface can be used to use simulator as a standalone module to start keys exchange and change settings.

### Web Interface

Web interface is accessed through http requests and it is used by other instances of the same module (e.g. one instance represents Alice and the other Bob). It must be run before any key exchange request from the module interface is raised so that web server is ready to receive requests when needed. Once running, it should not be stopped until the whole simulation ends. Web interface also allows to start a key exchange from a remote module if the simulator instance runs without higher level modules.

The interface basically contains all the functions required to perform a key exchange, both for quantum and classical channel and it is described by table 8.1.

<i>Web interface</i>		
Method Name	URL	Access Method
<i>BB84 interface</i>		
getQuantumKey	http://{simulator_IP}/sendRegister	POST
compareBasis	http://{simulator_IP}/compareBasis	POST
verifyKey	http://{simulator_IP}/verifyKey	POST
<i>E91 interface</i>		
getBasis	http://{simulator_IP}/getBasis	GET
setKey	http://{simulator_IP}/setKey	POST
<i>General interface</i>		
startKeyExchange	http://{simulator_IP}/startKeyExchange	POST
settings	http://{simulator_IP}/settings	POST

Table 8.1.

**getQuantumKey** is the method representing the quantum channel for BB84 protocol. It is the first method to be called when a quantum key exchange is requested. It receives the quantum bits measured by the sender, selects a random set of basis and uses them to perform measurements on the received qubits. If no errors occur during these phases, the call returns a 200 status code without response body to signal the sender it is ready for the next phase.

**compareBasis** method, as the name suggests, regards the basis comparison over the classical channel. It receives as body of the request the basis table used in measurements by the sender and, since the classical channel must be authenticated, the signature of the basis table with the private key of the sender. Inside this function key sifting is performed: after the signature has been verified the received basis table is compared with the one used locally and, when a difference in the tables is met, that bit of the key is discarded. The reply to this call is the local basis table. In such a way the caller can perform the same key sifting procedure on his side. Also in this case the returned basis table is sent together with a signature to let caller check the authenticity of the received data.

**verifyKey** represents the last stage of the BB84 protocol, performed again over the classical channel. The request body is composed of  $\frac{1}{3}$  of the exchanged key and an array representing the indexes of the bits sent as first parameter. Also these two parameters are signed with private key of sender to ensure authentication of the channel. Received bits are compared with the local copy of the key to check for the error rate. After this, compared bits are removed from the final key. The response to this method is the same subset of  $\frac{1}{3}$  of the key, this time extracted from the local copy of the key, so that caller can make the same checks on his side. Together with the subset of the key, the reply also contains a signature of returned data for the same reasons explained above.

**getBasis** is the method called from channel to require basis table to be used during measurements in E91 protocol. When called, simulator randomly generates a basis table by choosing

the requested number of basis from the list of available for this protocol and returns them to the channel to let it perform measurements.

**setKey** method is used by the channel to return calculated key and CHSH correlation value to the simulator as ending part of E91 protocol. As said, all calculation are made inside channel, hence in this method returned results are simply saved to let them be available for higher level modules that request them.

**startKeyExchange** is the method used to require a new quantum key exchange. If simulator is not integrated in one of the higher level modules that will be described by next chapter, it is possible to run it as a standalone module. In this case key exchange may be required through this method. With parameters passed in the body of the request, it is possible to specify the destination to exchange the key with, length of the requested key and the protocol that should be used to exchange the key. This method will call the above methods of the web interface of destination in order to complete the key exchange.

**settings** method is used to change settings inside simulator. Specifically this method allows to select the authentication method over the classical channel that should be used for subsequent exchange (*SPHINCS+* or AES-GCM) and the length of the chunks the generated qubits must be split into when performing the exchange. Qubits are spit in chunk since it is not possible to process a number of qubits of the same length the final key should have, mainly because of memory issue. Length of chunks can impact on execution time, hence also this parameter is editable. Currently only these two parameters are configurable, but the method has been thought to be simply customizable in order to easily add new settings.

## Module Interface

Simulator can also being imported in other program as any other python module. In this case simulator exposes an object-oriented interface that can be used to request quantum key exchanges. Before this functionality is requested, the web server interface of the module must be launched for both this and the target instance in order to make sure that all requests will be correctly managed.

Module interface expose a class that can be instantiated from high level modules. This class, called Simulator, inherits from an abstract class named *QKD*, that exposes basic methods to require a key exchange. The need to define an abstract class arises from the requirement of having abstract physical implementation from the logical one. In particular, if different QKD simulators (such as different real devices) implement the same interface, the high level module does not need to take care about the specific implementation of such simulators. In such a way it is possible for a higher level module to select and use different simulators (or real devices) transparently. The abstract interface can however be extended with other functions in order to meet additional requirements that a specific implementation may have. Methods of abstract class *QKD* are shown in figure 8.5.

---

```
class QKD:
    def begin(self, serverPort)
    def exchangeKey(self, key_length, destination, protocol, timeout)
    def end(self)
```

---

Figure 8.5. *QKD* abstract class interface.

Please note that, even if input parameters have been defined, thanks to the python flexibility the abstract class does not define returning values and the implementing class can select the returning values it prefers. At the same way, if one of the input parameter is not required by a specific implementation, it can simply be ignored.

**begin()** function is meant to initialise QKD simulator. In a real device this function can be used to prepare and synchronise quantum channel for the following exchanges. In BB84 implementation, this function is used to start the web server in a dedicated thread and set up the log module that will log any received request on file. It takes a port as argument that corresponds to the port used from the server and does not return values.

**exchangeKey()** function is the method to be called to start a quantum key exchange. Its input parameters are:

- *key.length*: it is used to specify the length the requested key must have. The real exchanged key will be  $\text{key\_length} + (\frac{1}{3} \times \text{key\_length})$  in order to use the exceeding length of the key during verification process.
- *destination*: it is used to specify the target instance to exchange the key with. In this implementation of simulator, this parameter represents the IP address and the port of the target web server and will be used to access both classical and quantum channels.
- *protocol*: this parameter allows to select the protocol that will be employed during the exchange. Currently available protocols are BB84 and E91.
- *timeout*: it is used to specify a time within the which the whole key exchange process must be completed. If the timeout elapses before the key exchange takes place the process is stopped and no key is returned.

In class implementation this method returns the exchanged key together with a boolean value that indicates whether the key has been successfully exchanged or not. If the key exchange fails before the key is exchanged (e.g. timeout elapses) the returned key is a python `None` value with the boolean value set to `False`.

**end()** function is intended to dispose QKD simulator. In a real device this function may be used to release used resources and let it be available for other modules. In simulator implementation instead, this function is used to stop thread running the web server and release software resources.

## Manager interface

This interface is implemented in simulator manager block shown in figure 8.1. The scope of this module is to provide an easy to use interface to access simulator functionalities. Hence, simulator manager is just another python module that collects all possible operations that can be performed on the simulator and exposes them with a user friendly interface.

Manager interface is realised through Flask web server. In order to create an intuitive interface, *Flask Dashboard Black* <sup>1</sup> has been used. This tool allows to generate a graphical interface that will be served by Flask web server. In this way users can select among the different operations by just clicking on the interface.

Proposed manager interface can be seen in figure 8.6. It basically consists of 3 panels, each one controlling specific functionalities. The first one allows to start a new key exchange, by selecting desired destination, key length and protocol to be used by using related controls. Another panel allows to modify simulator settings and currently consist of two controls to modify the chunk length of exchanged qubits and the authentication method to be used in classical channel (*SPHINCS+* or AES-GCM). Last panel allows to modify channel settings in order to enable specific attacks over both quantum (intercept and resend attack) and classical (man in the middle attack) channels. Intercept and resend attack refers to the simplest attack in a quantum channel. Basically the channel measures qubits before forwarding them to destination trying to gain information about their quantum state. Man in the middle attack refers to attack on classical channel. This attack has been currently foreseen, but it has not been implemented and is left to further implementations.

---

<sup>1</sup><https://appseed.us/admin-dashboards/flask-dashboard-black?ref=cm>

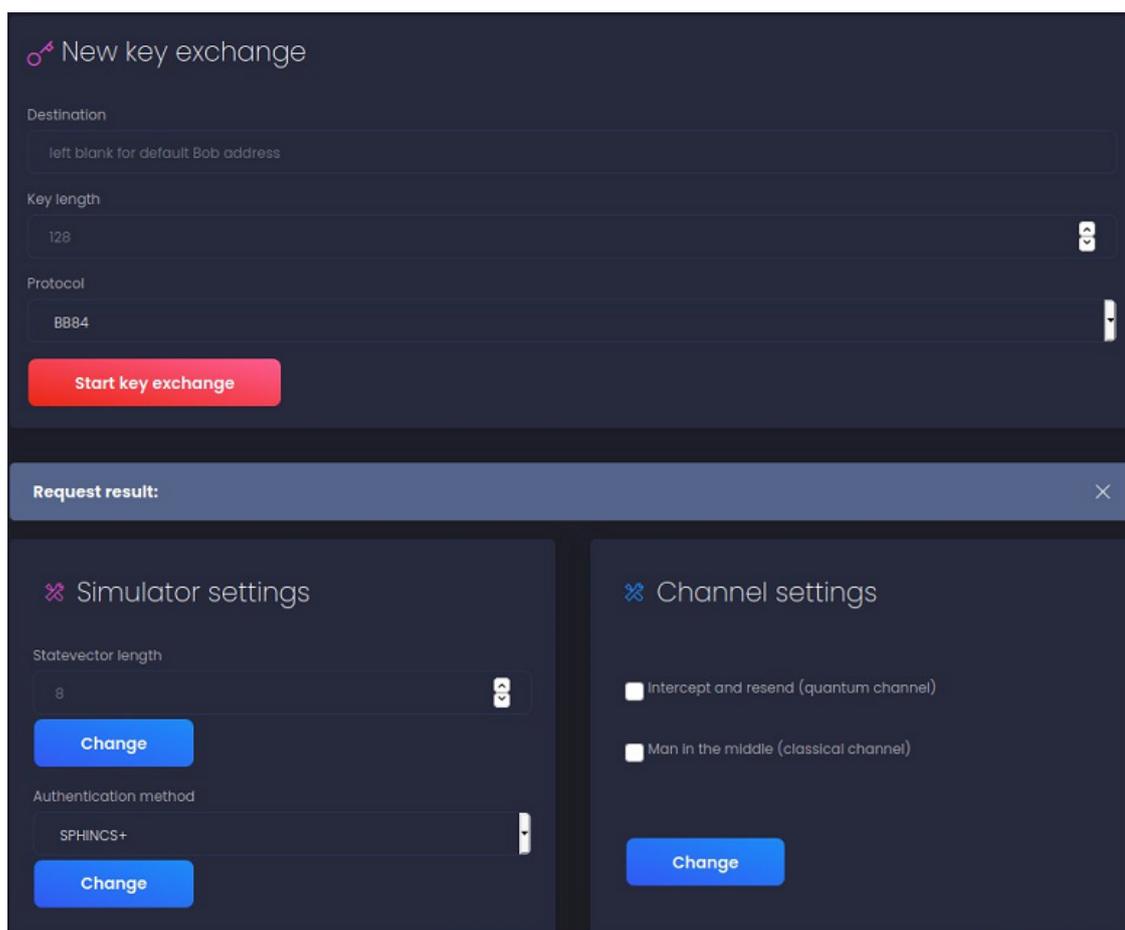


Figure 8.6. Simulator Manager Interface.

## 8.4 Criticalities

Even if the presented solution is capable of correctly simulate a quantum key exchange, it presents some flaws that will now be outlined.

The first criticality arises from the same considerations did for E91: if a simulated quantum register is serialised and sent through a web interface, its polarisation is lost. This happens because, in the simulated environment, also quantum data are sent over the classical channel. This means that quantum mechanic properties are not preserved. If for E91 this is a blocking issue due to the impossibility of simulate entanglement, in BB84 to overcome this problem qiskit offers the possibility to use *Statevector* objects. These objects allow to save the status of a quantum register in order to use it among different circuits. Hence, every time qubits need to be sent between two distant instances, these are saved in a *Statevector* that is serialised and sent to the other instance to allow it to perform its measurements.

Another problem regards the chunks the qubits are split into in order to achieve the desired key length. As said, qubits must be processed in chunks in order to avoid memory errors. The problem is that chunk length is limited to 14 qubits, after which memory errors are thrown. This is quite limiting since usually length of exchanged keys can be much greater, such as 256, 1024 and so on. This leads to a slow execution, since for 1024 bits of key at least 80 chunks must be exchanged (actually the number of chunks is greater since qubits may be discarded if they are measured in different basis). This limitation is due to qiskit framework and, up today, there is no way to overcome it.

A security concern regards the management of post quantum cryptography keys used to authenticate classical channel in case *SPHINCS+* is used as authentication method. Right now

each instance of the simulator owns a copy of its private key together with all public keys of other instances directly embedded in the source code. This solution is not scalable: if a new instance must be added, all previous instances must be manually updated.

Besides, communication between modules and storages does not run over https. All exchanged data (keys also) pass through a not secure connection and are thus available for anyone trying to intercept them.

The criticalities outlined here have been considered during design phase, however they have not been fixed mainly because of the role the simulator has in this work. The goal of this work is to present a QKD system that can be used in softwarised network. This means that the solution should be independent by the physical realization of the QKD system itself. This is also why a abstract interface has been defined as module interface: in this way simulator can be easily changed with other implementations, either physical or simulated. The role of the implemented simulator is just a placeholder for the higher level modules that will hopefully use real devices when they will be available. Its scope is limited to helping to outline the needs of higher level modules in order to design a reliable solution, but it should not be considered as a landmark for other QKD implementation. After these considerations it has been decided to not fix all security related issues, since they will be managed by the real implementation of a QKD device when it will be used in a real application.

For the same reason, BB84 simulator ends key exchange with key verify procedure. No error correction or privacy amplification procedures are not employed. However these procedures can still be useful as study case and are left to be implemented in future works.

At the same way since this is a simulation, no channel errors occur. This implies a QBER value always equals to 0% unless an eavesdropper disturbs the communication. Hence QBER threshold in this implementation is not the 11% value the literature suggests. It is instead set to 0%, hence any error on the key will cause the key to be discarded because any error on the key can be addressed only to an eavesdropper. This can be modified in future implementation by modelling a simulated channel error and setting again the QBER threshold to 11%.

## 8.5 Workflow

In this section examples of key exchange using the proposed simulator are presented. Regardless of the protocol the simulator will use, a generic example of workflow involving simulator components can be seen in figure 8.7. At high level, the steps performed by this implementation are the

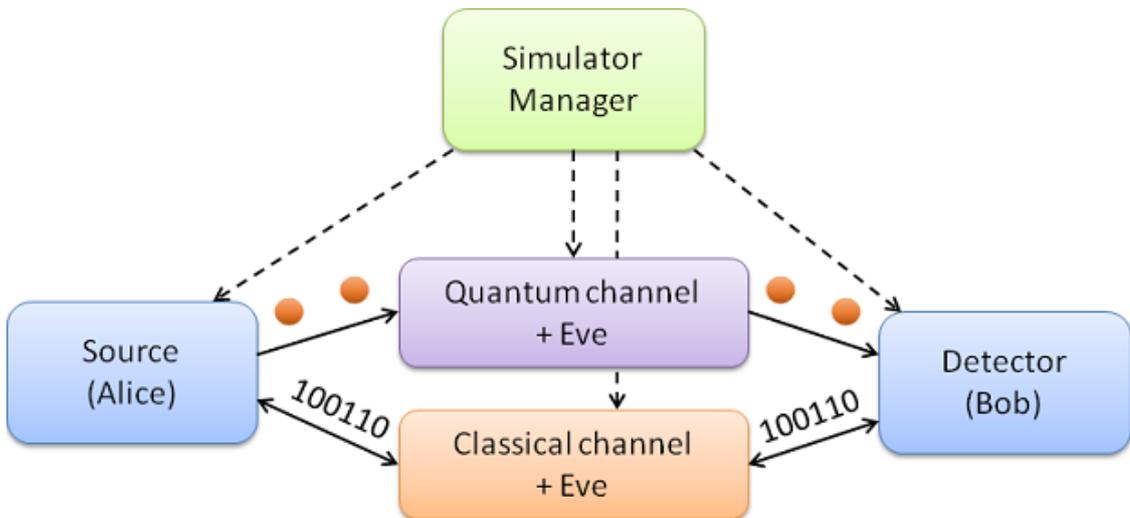


Figure 8.7. Simulator Generic workflow.

following:

- Simulator manager is used to tune available settings on the other components. These settings may refer to attacks on channels as well as specific protocol settings.
- Simulator can be used to require a new key exchange. It will forward the request to source interface (Alice) and the exchange will actually start.
- Source generates and sends qubits over the quantum channel.
- Channel may interfere or not with the communication depending on global settings. After any operation it can perform on the data, it forwards them to the destination (Bob).
- Destination performs its operation on the received qubits accordingly to the chosen protocol.
- Source and destination performs further operations on the classical channel in order to realise key sifting, error correction, privacy amplification and any other operation a specific protocol may require. Even in this case, all information exchanged between source and destination are forwarded through the channel that has full control over the data.
- When key has been exchange, it is returned to the calling application.

More detailed workflow can be seen by considering specific protocols implementation. Here the two implemented protocol will be considered and a dedicated workflow will be provided.

## BB84

In this example the key will be requested by a high level module that requires a BB84 exchange with post quantum authentication. The main interactions between different instances as well as how high level modules can access the simulator will be shown.

A key exchange starts when a high level module requires it. In order for a high level module to request operations to simulator, it has to retrieve the object representing the simulator module interface. Once the object has been retrieved it can issue commands to the simulator by using functions shown in figure 8.5.

The instance of simulator which receives the request must access the web interface of the other instance in order to start the key exchange. Thus, web interface must be up and running before any request is issued. This is done in the initialisation phase from the high level module by calling `begin()` function. Both instances must call this function in order to allow both of them to start a key exchange. For simplicity the simulator instances involved in the exchange will again have the name of Alice and Bob.

Once everything is ready, the high level module of Alice requires a key exchange by calling `exchangeKey()` function. The simulator instance starts the key exchange with the target simulator by performing a sequence of steps as can be seen in figure 8.8.

In particular, inside `exchangeKey()` function, Alice simulator perform the following steps:

- It checks if in its vault storage is already present a key exchanged with desired destination. If there is, it returns this key, deletes it from storage and return, else the key exchange starts.
- It computes  $\frac{1}{3}$  of required key length and adds this quantity to the final key length in order to return a key of required length and discard the exceeding bits during verification phase.
- It computes the length of qubits to be exchanged starting from desired key length: statistically, since basis choice is made between two possible basis, half of the qubits will be discarded during key sifting procedure. Hence qubits length will be key length multiplied by two, plus a 15% more of length to ensure the desired length is reached.
- It split qubits length in different chunks (whose default length is 8 qubits) and for each chunk:

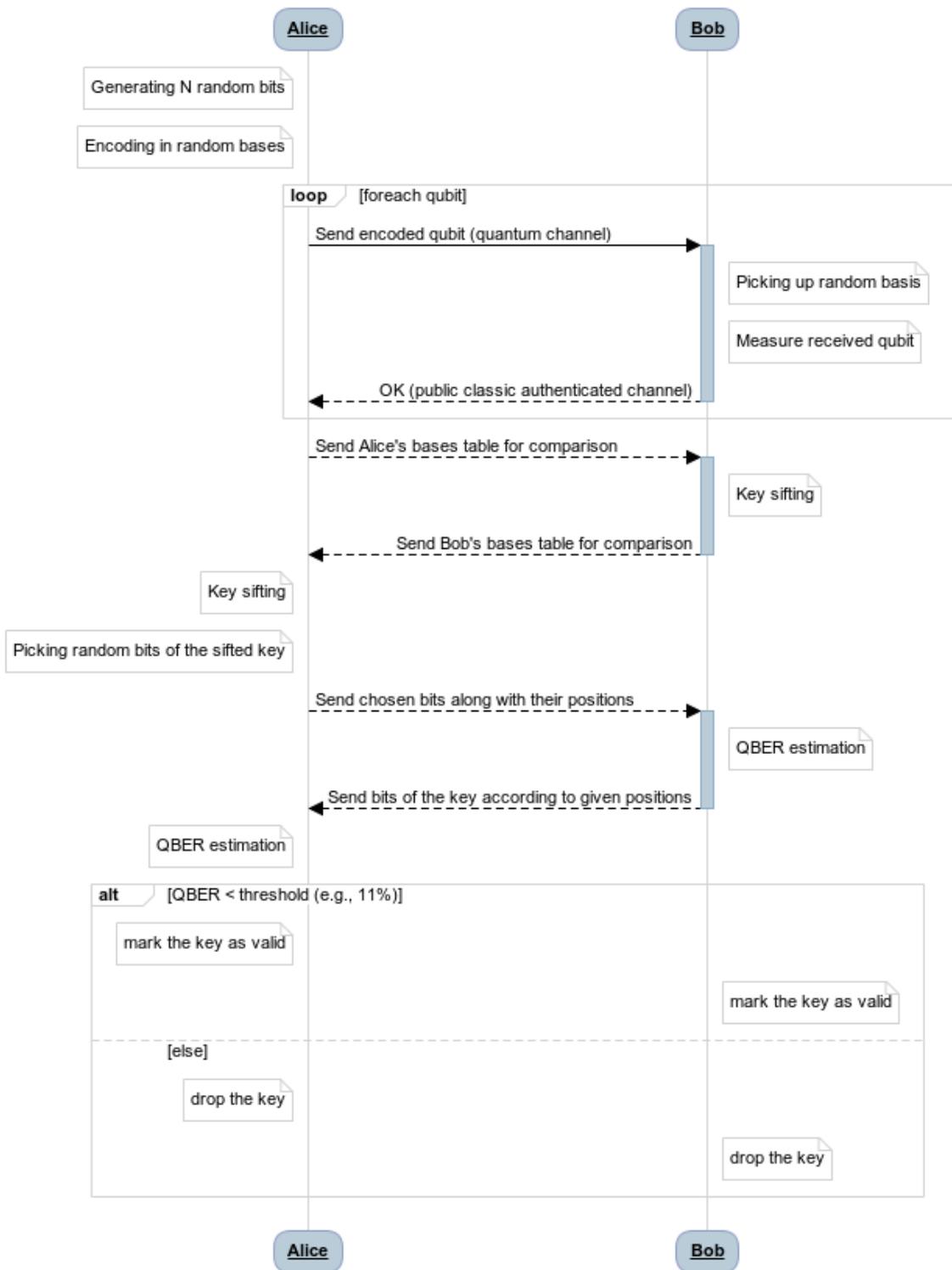


Figure 8.8. BB84 simulator workflow.

- By using the true random number generator module, it generates a bits string of the same length of the chunk.
- By using a qiskit circuit it encodes the bits string in a statevector object by using a set of randomly chosen basis. The used basis are saved in a basis table for further processing.

- It serialises statevector objects and sends them to Bob simulator. The Statevector is sent to Bob simulator using `http://bobIP/sendRegister` endpoint.
- It checks the status code of the previous call. If it is '200' it knows Bob simulator has randomly selected a set of basis and measured received Statevector in those basis by using a qiskit circuit, else an error occurred and the function ends without returning a key.
- It signs the basis table formerly saved with its private key and send it to Bob simulator using `http://bobIP/compareBasis` endpoint. This call returns the Bob simulator basis table.
- It check signature on received basis table using Bob public key. If signature is not valid it ends the function and does not return a key.
- It compares the received basis table from the previous call with its local copy in order to sift the key: if a different basis is chosen for a particular qubit, the related bit is discarded. Note that Bob simulator performed this step during the previous http call before returning its basis table.
- It randomly selects a set of bits along the key that will be used to calculate QBER value and save the indexes of these bits in a separated array.
- It signs selected bits and indexes array with its private key and sends them together with their signatures to Bob simulator by using `http://bobIP/verifyKey` endpoint.
- It receives back Bob's bits of the key at the same position of those sent together with a signature for those data. If signature is not valid the function ends without returning a key.
- It compares received bits with the one sent and checks the error rate as number of equals bits divided by key length. Since this is just a simulation and channel errors are not possible this value will likely be equal to one (which means the QBER is 0%). Hence if value is different from one the key is discarded, otherwise the key is verified and successfully returned to high level module. Bob simulator performs the same step before returning its basis table and saves the key in its mysql database waiting that its high level module will request it.

## E91

An example of workflow can be seen for E91 as well. As said, E91 protocol involves a new party that is in charge of providing entangled qubits to Alice and Bob. Besides, communication channel plays a crucial role in this case, since measurements are performed here due to the impossibility of simulating entanglement behaviour. As per BB84, E91 key exchange starts with a call to `exchangeKey()` function to Alice interface. Inside this function the following operations are performed:

- Alice checks if in its vault storage is already present a key exchanged with desired destination. If there is, it returns this key, deletes it from storage and return, else the key exchange starts.
- Alice request a new key exchange to channel by using `http://ChannelIP/startE91exchange` endpoint.
- Channel now perform a set of operations that can be seen in figure 8.9 and are here described.
- Channel requires entangled qubits to singlet source by using `http://SSourceIP/getQubits` endpoint.
- Singlet source replies with the required singles which were already be prepared beforehand. If more qubits than available are requested, it produces the remaining qubits within the request in order to reply with the correct number of qubit pairs.
- Channel requires to Alice a basis table to be used during measurements by using the endpoint `http://AliceIP/getBasis`.
- Alice uses its random number generator to create a basis table and returns it to channel.

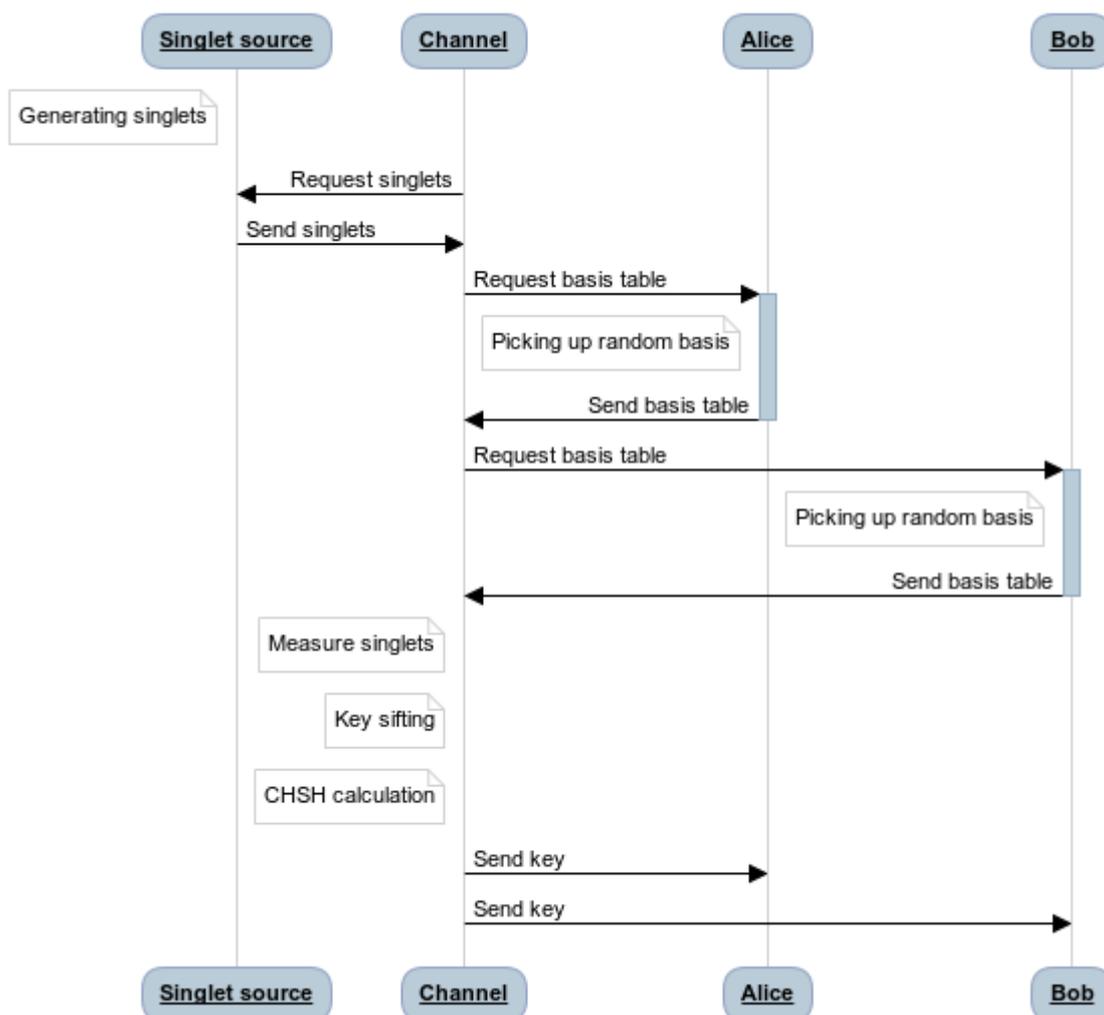


Figure 8.9. E91 simulator workflow.

- Channel requires to Bob a basis table to be used during measurements by using the endpoint `http://BobIP/getBasis`.
- Bob uses its random number generator to create a basis table and returns it to channel.
- Channel performs measurements by using basis chosen by Alice, then it performs another run of measurements by using basis chosen by Bob.
- Channel performs key sifting by cancel out bits of the key in which basis chosen by Alice and Bob is different.
- Channel performs CHSH calculation to make sure entanglement relationship on measured qubits is valid. This is done by checking that CHSH value is about  $2\sqrt{2}$ .
- Channel returns generated key to Alice and Bob using `http://AliceIP(BobIP)/setKey` endpoint so that they now share the same key.

## Chapter 9

# QKD Module (QKDM)

In this section the proposed implementation for a QKD Module (QKDM) will be presented, as defined by ETSI GS QKD 004 (Application Interface) standard [53]. Its architecture and interface will be explained, focusing on the interaction between this module and the underlying level (the QKD simulator).

### 9.1 QKD Module Design and Architecture

ETSI GS QKD 004 defines the interface a *QKD Key Manager* should present in order to represent a QKD Module that can be used both by a high level security service (IPsec, TLS and so on) or a REST-based key server.

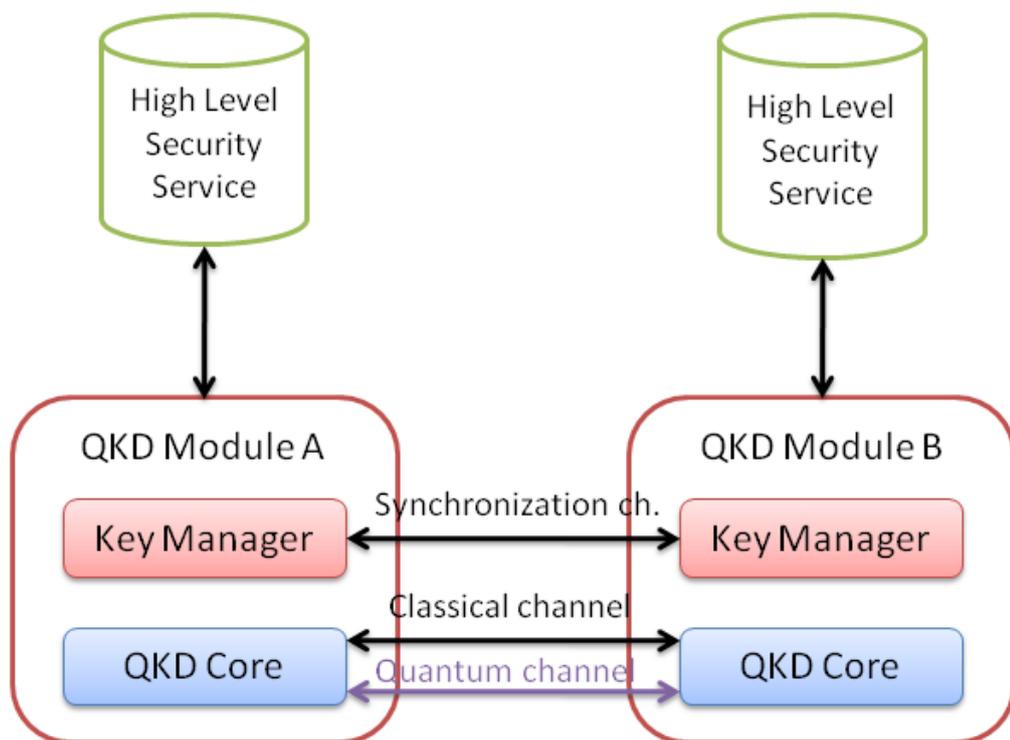


Figure 9.1. Conceptual scheme of QKD Module.

As can be seen from figure 4.1, standard requires that QKDM exposes an interface to the high level application in order to let it require QKD operations. Besides, it internally must be able to

control a QKD device to realise such operations and it must own a synchronisation interface over a classical channel to synchronise QKD operations with the target module. Hence the module comes with two interfaces: a module interface for high level interaction and a web interface for module-to-module synchronisation. In figure 9.1 is shown the basic building blocks and their main interactions of a QKD module.

QKD module is basically composed of two main blocks:

- QKD Core. This block represents the device that actually perform key exchange. In this implementation this is realised with the QKD simulator presented in previous chapter. However, the modular implementation proposed allows to simply manage different QKD cores, in order to use other implementation or even real devices without requiring changes to QKD module implementation.
- Key Manager. This block represents the real implementation of ETSI standard. It provides an interface to higher level modules, talks to other modules to synchronise key exchange and, if an association with another module is established, it continuously exchanges keys with peer QKDM exploiting the QKD core.

This implementation is hence aimed to create the interfaces needed to let system work and manage the internal QKD core. Please note that standard defines only the module interface towards the high level module (QKD API) and does not impose anything related to synchronisation interface nor internal implementation choices. In order to realise the required functionalities QKD module has been developed with a set of components as depicted by figure 9.2.

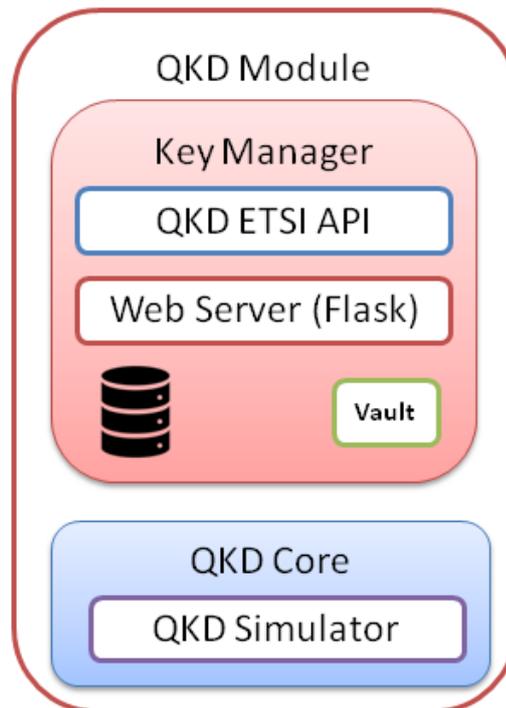


Figure 9.2. QKD Module internal components.

As can be seen from the figure, the block aimed to implement QKD Core functionality is QKD Simulator already developed. However, some modification to the simulator presented in the previous chapter has been made. Ideally, the QKD simulator presented in chapter 8 can be used as is for this development. The idea is that any device (either simulated or real) inherits from an abstract python class called *QKD*. In this way all devices are capable of provide the same basic interface, that then can be extended if needed. QKD module can hence make sure that it will always be able to interact with any QKD device by exploiting the simple interface

it provides, without taking care of possible changes in the underlying implementation. Hence actually no modification to the QKD simulator is needed. However, two different needs emerged while developing QKD simulator. In order to simulate a QKD key exchange for the QKDM, a simple simulator can be used. The goal of this project is to integrate a QKD system in a distributed environment, outlining the main challenges. As already said, this module should not take care about the underlying implementation and how it is realised. Hence a very simple simulator can be used: even a fake key exchange can be set up for this purpose. However, while developing the QKD simulator the needs of study different scenarios came up. This leads to the realisation of a modular QKD simulator as presented in chapter 8, that can easily be extended by adding mathematical models of physical phenomena that may require attention. Even if this simulator can be used as is, it has been chosen to simplify its implementation in order to focus on the higher levels only. Hence, the simulator used here is a lightweight version of the one already presented. It uses BB84 protocol or exchange a fake keys without using a QKD protocol at all: this allows to remove simulator manager and singlet source block. Besides, if the fake key exchange is requested, the simulator does not need to use all that additional component that can be seen in figure 8.2, such as storage and logging functionality: it can just forward randomly generated key through a web server. This significantly reduces the time required by a key exchange and allows to stress the system with a higher number of keys to be managed. The choice between BB84 algorithm and the fake key exchanger can be done by modifying a setting in the QKD Module and this choice cannot be changed at runtime.

If QKD simulator is the component aimed to implement QKD Core functionality, all other modules shown in figure 9.2 are aimed to implement Key Manager functionalities.

ETSI API and web server blocks represents the two interface needed to interact with QKDM and will be explained in the next paragraph.

The proposed implementation makes use of a mysql database in order to save information about the currently exchanged keys together with their handles that will then be returned to high level modules. However, keys are not saved inside mysql database. In order to ensure that keys are always in a safe storage *Vault* service is used <sup>1</sup>. Vault is a shared datacenter that allows to securely store any secret or sensitive data and can be accessed through HTTP API. This allows to increase security of the overall system since, at this level, security of QKD is no longer entrusted to quantum mechanics but to the way keys are managed in classical systems. Hence, while mysql database stores, for a given key handle, information that allows to understand if the exchange is still in progress, if it is completed and so on, in Vault storage a copy of the key is saved together with its key handle (key handle is the entry point to easily recognise different keys). Key handle can be safely stored in mysql too since, as standard points out, it is not related to the key and there is no possibility to gain information about the key starting from its handle.

Another security concern that has been investigated for this implementation is the security on the web interface used for synchronisation purpose. In the underlying level has been seen how a simple authentication on the classical channel is enough to consider the channel secure against third party interventions. In this case, modules need to exchange data about the handles in order to correctly synchronise desired exchanges. If a man in the middle intercept the communication, he cannot gain information about the keys (that are never exchanged within this channel), but he can change information about the key handles representing the currently exchanged keys. Thus, web interface needs to be secured. The same considerations did for securing classical channel in QKD Simulator apply in this case. Thanks to the capability of the system to produce keys a preshared keys system can be used to secure communication on this layer. In such a scenario part of the produced keys can be used to feed preshared key storage when keys need to be refreshed.

## 9.2 Interfaces

As already said, also this implementation is composed of two different interfaces:

---

<sup>1</sup><https://www.vaultproject.io/>

- Module interface: API defined by ETSI standard.
- Web interface: used by other instances of QKD module to synchronise key exchange.

### Module Interface

Module interface implements functions defined in ETSI standard, shown in figure 4.2. These functions are collected under a class named *QKDMModule* in order to be easily accessed by higher level modules. Basically, implemented functions are:

- *OPEN\_CONNECT(source, destination, QOS, Key\_stream\_ID, status)*. This function is used to open an association between the two QKD modules that need to exchange the keys. It should no return until the association has been established or timeout (specified in QOS parameter) occurs. Internally, it creates a key handle if *Key\_stream\_ID* input parameter is set to NULL. Key handle is generated by using *uuid* python module in order to have it in a UUID version 4 format as specified from standard. When the *Key\_stream\_ID* is generated it is sent to the target QKD module through the web interface in order to create the association. If the remote call is fine, the handle is saved inside the database in order to be easily retrieved in other functions for further computations. This function returns *Key\_stream\_ID* together with status parameter that is filled with a value indicating whether the function execution incurred in errors or not. If *Key\_stream\_ID* parameter is not NULL, an internal check on database is performed in order to understand if the handle refers to one opened from the peer QKD module or it is a user defined handle. In this latter case a further check is performed to make sure the selected handle is unique, if this requirement is not meet an error is returned. Once the association between two modules has been established, they will start to exchange keys in order to maximise key rate.
- *GET\_KEY(Key\_stream\_ID, index, Key\_buffer, Metadata, status)*. This function is used to retrieve a key associated to the handle (*Key\_stream\_ID*) passed as parameter. Since modules continuously exchange keys for a given key handle, two high level module can request the same key by using index parameter, that refers to a particular key in the stream. This function returns the required key together with status parameter that is filled with a value indicating whether the function execution incurred in errors or not.
- *CLOSE(Key\_stream\_ID, status)*. This function terminates the association related to the handle passed as first argument. This means that the two modules linked by this association will stop generating keys.

Status parameter has the same aim for all the three functions. Its value indicate if the function correctly succeeded or an error occurred. Its meaning can be seen in table 4.2.

Besides the *QKDMModule* class, a copy of these functions has been also implemented in an http interface in order to access the *QKDMModule* even if the device is not physically attached to the requesting machine.

### Web Interface

As for simulator, web interface for QKDM is realised with python module *Flask* for the same consideration of simplicity of use already discussed. This interface was not specified in ETSI standard and has been designed to meet the implementation requirements faced.

The designed interface is relatively simple and is composed of the functions shown by table 9.1.

**open** method is called inside *OPEN\_CONNECT* function to establish an association for a given *Key\_stream\_ID*. This function receives the *Key\_stream\_ID* as body parameter together with source URI identifier and quality of service information. In this way it can store information related to this stream of keys in its internal mysql database and use it to manage keys exchange. Basically this function performs a check on input parameters and, if everything is fine, store

Web interface		
Method Name	URL	Access Method
open	http://{QKDM_IP}/open	POST
start	http://{QKDM_IP}/start	POST

Table 9.1.

received information. It returns status code 200 if no error on input parameters is found to signal peer QKDM that association has been established. In case of any error a status code of 400 is returned and the peer QKDM will notify the error to the caller function.

**start** method is used to synchronise a key exchange. It is called inside the thread that constantly exchange keys for each stream ID. It is used to synchronise the single key exchange inside the stream. As soon as a QKDM wants to start a key exchange, it calls start method sending the `Key_stream_ID` and the index as parameters. Inside this function a table in mysql database named `currentExchange` is written with the just received parameters and the key exchange is allowed to start. In this way the two QKDM can agree about the key to be returned if a specific index is requested with `GET_KEY` function by an high level module.

### 9.3 Criticalities

This implementation suffers of some criticalities that will now be discussed.

The first note on this implementation regards Quality Of Service (QOS). ETSI standard defines QOS parameter as a structure containing the following fields:

- *Key\_chunk\_size*
- *Max\_bps*
- *Min\_bps*
- *Jitter*
- *Priority*
- *Timeout*
- *Time to Live*
- *Metadata*

`Key_chunk_size` is the length of the key returned by a call to `GET_KEY` function. This parameter is correctly managed in this implementation as well as `Timeout` parameter that is used to stop function execution if it exceeds time specified. All other parameters have not been managed. Hence it is not possible to fully define quality of service parameters are required from standard. The implementation of these parameters is left to further developments of this work.

Another criticality regards a security assumption on the QKDM itself. The module is considered to be in a trusted node. This means that, while communication with other modules requires a certain level of security, communication between QKDM and underlying level is not secured at all. Keys are stored in a secure storage (Vault) when they need to be processed at a different time from exchange, but when they are actually managed (when they are in RAM memory) they are fully accessible within the system. There is no secure memory region that allow a secure management of the key (i.e. Intel SGX <sup>2</sup>) from this application only.

---

<sup>2</sup><https://www.intel.it/content/www/it/it/architecture-and-technology/software-guard-extensions.html>

Concept of trusted node can be found in ETSI standard too. However, this is a weak security assumption and can represent a criticality in the standard itself. It would be better that security does not rely on any assumption at all in order to build a device independent QKD system.

One last observation concerns connection between QKDM and the underlying QKD implementation. Right now, QKD device instance is attached by including the python simulator module inside QKDMModule. The import was renamed in such a way that, inside the code, no reference to simulator is found. Specifically, it is included as follows:

```
from Simulator import Simulator as QKDCore
```

In this way the object allowing the key exchange will be referenced as *QKDCore* rather than *Simulator*. If it is needed to change the underlying implementation it is enough to change the imported module with the desired one and rename it as *QKDCore*. In this way there is no need to change any other detail in the implementation and the QKDM will continue to work normally. Since simulator is only made by software (it is a python module), it is always available when imported. If a real device needs to be used, it is possible that it is not available due to a temporary physical detach from the system, a malfunctioning or any other reason. In this case it is possible to implement a check of the correct physical device communication within the constructor of *QKDCore* class or within the `begin` function. In such a way it can return an error to the QKDMModule that can correctly quit the application. However, when real devices will be employed it is possible to modify the attachment procedure in order to meet specific requirements that cannot be foreseen in this phase.

## 9.4 Workflow

An example of use of the QKD Module will now be shown. QKDM can be used by a secure application entity as well as a *Key Server* compliant to ETSI standard (that will be presented in the next chapter). This distinction will not affect QKDM, that will behave at the same way independently by the high level module.

Before starting any operation on the QKDM, its web interface must be launched. The web interface must be launched by starting QKDMModule from a terminal. Hence, python QKDMModule can be imported in high level module in order to access its module interface, but it can also be launched as a standalone program to run its web interface. When run in a terminal, QKDMModule accepts an input argument to specify the port number the web interface must be accessible to. Then, before starting the web interface, it checks for the underlying QKD system in order to make sure it is a class which inherits from the abstract class *QKD* (this is done to make sure that used QKD system presents the interface the module is expecting and no runtime error will be thrown for unknown software calls). If the class is not an instance of *QKD* the program stops its execution, otherwise it calls `begin` function on the QKD device and starts its web interface.

Once the web interface of all involved QKDM is correctly started it is possible to call functions that allow to exchange keys. Considering to have two high level applications, called *APPA* and *APPB*, that want to exchange a key using their QKD modules, called *QKDMA* and *QKDMB*, the following steps need to be performed:

- *APPA* calls `OPEN_CONNECT` function, specifying its URI, the destination URI, the desired quality of service and a `Key_stream_ID` set to `NULL`.
- Inside *QKDMA* input parameters are processed. Since `Key_stream_ID` is set to `NULL` a new handle (UUID version 4) is generated. This handle is sent to the target module together with quality of service information by calling `open` method on the *QKDMB* web interface.
- *QKDMB* receives a call on `open` method of its web interface. Inside it, it check if the received handle is already in use (handle must be unique) and eventually returns an error. If it is not used, it saves all received information in a mysql database and reply with 200 status code.

- If QKDMA receives a status code different from 200, it returns an error code of 4 (no QKD connection available) to the calling function. Otherwise it saves the input parameter together with the generated handle to its database and return the handle itself together with the status code 0 (successful) to the calling function.
- APPA receives the generated handle and forwards it to APPB. Please note that the way these information are exchanged is outside the scope of this implementation.
- APPB receives the handle from APPA and use it as `Key_stream_ID` when calling function `OPEN_CONNECT` on QKDMB. Note that it does not need to specify quality of service since these data have been forwarded from QKDMA.
- Inside QKDMB `OPEN_CONNECT` function input parameters are processed. Since parameter `Key_stream_ID` is not NULL it is searched inside mysql database to check if another QKDM forwarded information about this handle. In this case, QKDMB will found related information inside its database and will update the information in it to signal that `OPEN_CONNECT` function has been called on this side too for this handle. Then it replies with a successful status code to APPB.
- Once QKDMB receives `OPEN_CONNECT` function call with a stream ID already presents in its database, it knows that a link for that stream has been established on both QKDMs. Hence a thread inside QKDMB starts to exchange keys until a limit number specified in configuration phase is reached. Keys exchange consists of the repetition of the following steps:
  - the current number of exchanged keys is checked from database. If there is no more room for new keys the system waits that high level modules require some keys in order to free memory, otherwise a new key exchange is started.
  - QKDMB calls `start` method of the QKDMA web interface is called in order to signal that an exchange belonging to a given `Key_stream_ID` with a specific index is required. Inside this method QKDMA updates its `currentExchange` table in order to exchange requested key and get ready for `GET_KEY` requests that may occur on its interface. After that it returns 200 status code. This step is represented by number 1 in figure 9.3.
  - If QKDMB receives a status code of 200, synchronisation phase is considered complete and it saves the couple `Key_stream_ID`, index on its database as reference to the key that is going to be exchanged.
  - QKDMB calls `exchangeKey` function on the underlying QKD Core to actually starts key exchange. This operation is depicted by number 2 in figure 9.3 and causes the execution of the following steps:
    - \* BB84 implementation of QKD Core calls `sendRegister` function in order to send qubits to the pair instance (number 3 in figure 9.3) .
    - \* After sending qubits, simulator starts basis comparison by using `compareBasis` endpoint of simulator web interface (number 4 in figure 9.3). This operation corresponds to key sifting.
    - \* The last step is about verifying key exchange by comparing some bits of the key. This operation is done by using `verifyKey` endpoint of simulator web interface (number 5 in figure 9.3).
  - Key returned from `exchangeKey` function is saved in vault storage and the number of currently available keys is updated.
  - QKDMA calls `exchangeKey` function as well in order to retrieve the same key and update its storage consistently.
- In the meantime the high level module APPA calls `GET_KEY` on QKDMA.
- In `GET_KEY` function the requested parameters are checked. In particular `Key_stream_ID` requested must be related to the one associated to the calling APPA and a stream for

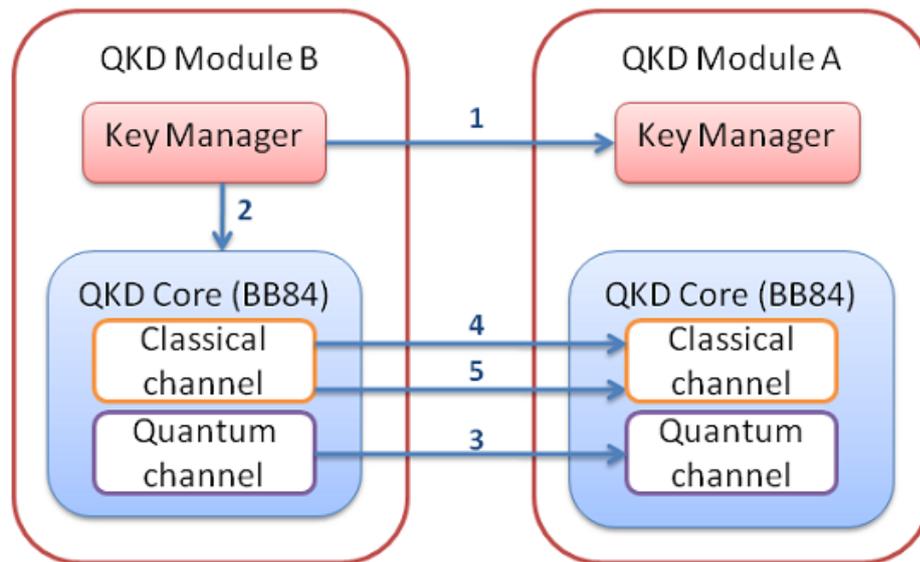


Figure 9.3. Key exchange workflow.

this ID must already have been established with the peer QKDMB. Then index parameter is checked. It must refer to an available key in the stream or otherwise an error will be returned. If input parameters are correct, QKDMA returns the selected key to APPA.

- Synchronisation of the index inside the stream for the two APPs is left to the APP implementation itself. At this point APPA can send a message to APPB to let it know the index of the key to request to its QKDM, else they can request keys with increasing indexes without a synchronisation phase. In all cases, this is out of the scope of this implementation and this decision is left to higher levels implementation.
- APPB calls GET\_KEY with the same Key\_stream\_ID and index and receives the same key. Now the two APPs can use this key for their purposes.
- APPA does not need further keys to be exchanged with APPB, so it calls CLOSE function specifying Key\_stream\_ID used so far.
- APPB calls CLOSE function from its QKDM to close the association.

# Chapter 10

## QKD Key Server

After having shown how a QKD device has been implemented by means of a simulator and how access to this device has been standardised by means of a QKD Module, in this chapter a solution to integrate QKD in softwarised networks is proposed. The designed solution is compliant to ETSI GS QKD 014 standard [54] in order to be easily adopted and integrated from existing solutions. ETSI standard has already been introduced in chapter 4. This chapter is aimed to explain in details how the defined REST-API has been implemented to realise a *Key Server*, outlining the main implementation choices.

A key server is a network node in a distributed infrastructure capable of sharing quantum keys with other key servers and return such keys to the requesting applications. Keys are shared by using QKD Module defined in the previous chapter. Basically a key server can manage more than one QKD Module in order to exchange keys with different destinations at the same time. Besides, since it is intended to work in a network, it should be able to serve multiple remote application.

### 10.1 Key Server Design and Architecture

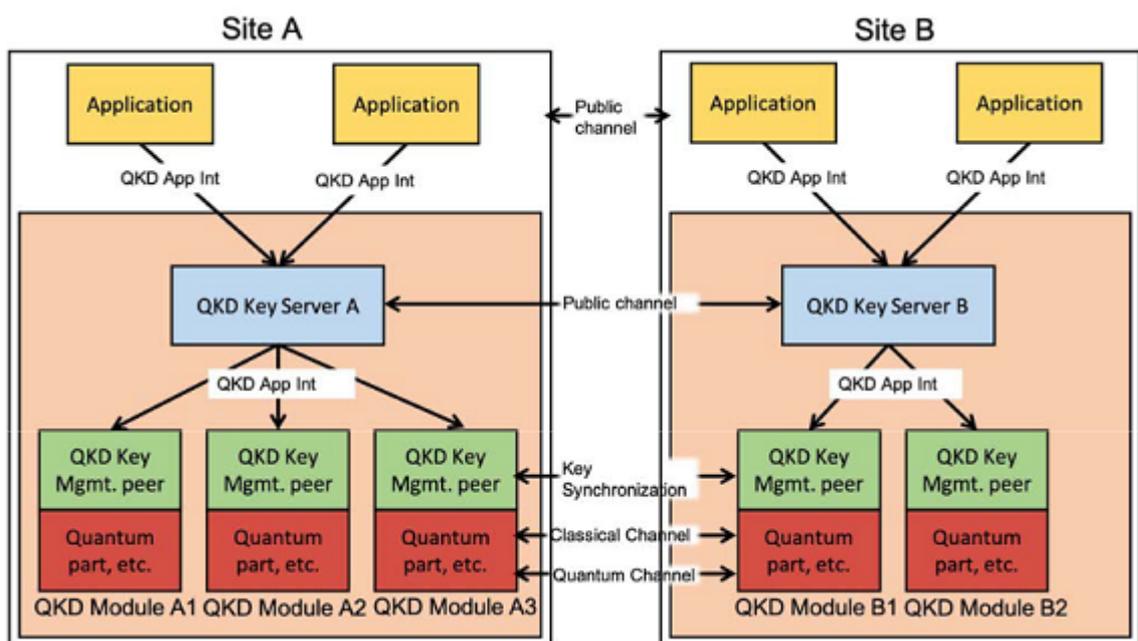


Figure 10.1. Structure and interaction of two QKD Key Servers. (source: ETSI GS QKD 004 [53]).

ETSI standard defines a set of REST-based API that a key server should expose, but it does not impose anything related to the internal implementation of the server itself, neither how it should communicate with other key servers. This API, specified by table 4.3, are part of the *northbound interface* designed in this implementation, that allows high level security application to access resources inside key server. Key server also presents a *southbound interface* for communicating with connected QKD modules. A block diagram of the presented solution outlining all its internal component is depicted in figure 10.2. As can be seen from this figure, key server architecture is composed of a number of different building blocks that will now be introduced.

Besides the exposed interfaces, proposed key server makes use of the following components:

- An *authentication* entity to allow access to the resources only to the authorised high level application.
- A *secure storage* to safely store exchanged keys.
- A *database* to save information related to keys or other useful data.
- A *core component* that manages all internal operations.

## Authentication

In ETSI standard is not unlikely to find references to security perimeters in which communications are considered safe without requiring particular precautions. This is true also for communication between key server and high level applications. However, in the case of a distributed environment, it is not wise consider security perimeters that includes final applications. It has been studied a way to secure communication between key server and high level applications in order to let this implementation really employable in a real context.

Communication between high level application and key server runs over https. This is required because key server returns quantum exchanged keys to high level application requests, thus communication must be encrypted. Besides it is necessary that only the authorised applications can access the key server, in order to make sure that untrustworthy applications do not get access to the keys.

Authentication in proposed key server is managed by a component named *Keycloak*<sup>1</sup>. Keycloak is an open source access manager that allows to authenticate applications and services. The advantage of Keycloak is that it is an out of the box independent system. Thus it is easy to integrate inside another application in order to have tested authentication features that receives constant updates against new threats. Basically, Keycloak is an external server that can be accessed independently by both the high level application and the key server. High level applications register to Keycloak in order to get permission to access key server. When they need to query for some keys, they request an *OpenID Connect* token to Keycloak. Keycloak releases the token to the registered application and sets a time to live to the released token, so that it will expire after a given amount of time. Applications use this token inside the https requests they make to key server. Token is inserted in the *Authorization* header of the https request. Key server uses the token in the request to query Keycloak about the authentication status of the requesting application. If Keycloak recognises the token as valid, key server process the application request, otherwise an error is returned. Do not pass a token in the request causes the immediate failure of the request itself. In such a way, all applications that try to access the server are correctly authenticated and authorised to access the server resources.

In this implementation Keycloak is managed through a docker container, in order to be easily accessible from all interested parties.

---

<sup>1</sup><https://www.keycloak.org/>

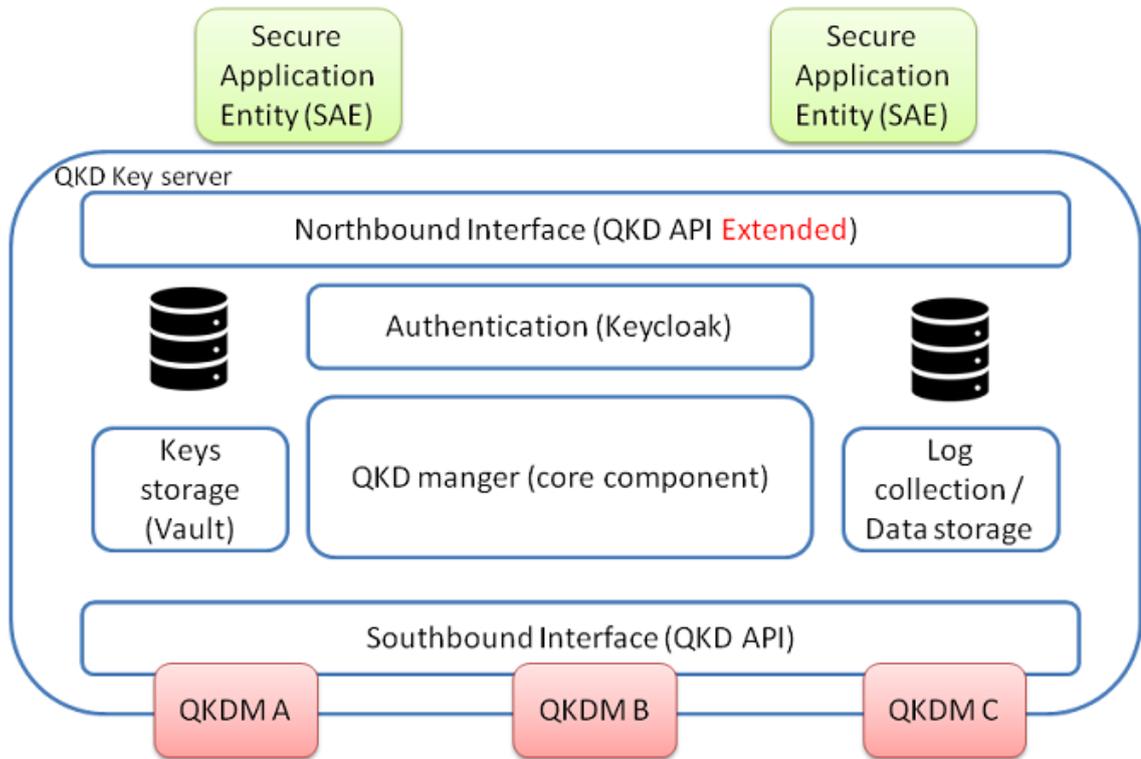


Figure 10.2. Key Server block diagram.

### Secure storage

The need of a secure storage has already been discussed in previous chapter, while talking about QKD Module. In a key server the desired behaviour consists of returning keys as soon as they are requested. Keys are requested to the QKDM directly connected after an association between it and another one is established. Hence, in key server there is not actually need of a secure storage since keys are just forwarded from the underlying level to the upper one.

However key server has been designed as a resources manager of the whole system. Since more than a QKDM can be available in the same key server, it has been thought of having a unique access point for secure storage, in which all modules can store information. Hence it is the key server that manages secure storage and provides an access point for this resource to the attached QKDMs during registering phase. In this way, only a secure storage server needs to be run and all modules can access to a dedicated memory space inside it, rather than set up different secure storage for each module.

Hence, even if key server does not make directly use of secure storage, it manages access to it and redirect QKDMs to the correct memory space to use. As already said, secure storage is realised by using Vault. Just like Keycloak, Vault has been integrated through a docker container for a quick deployment.

### Database

Another important building block for key server is represented by classical storage. Key server, as well as any normal server, is stateless. It does not allow to share variables among different http calls, thus a mysql database to store current status inside the server has been employed. Database allow to store information such as the current number of keys present in secure storage grouped for different key server peers, the number of attached QKD module and the URI of known high level applications that use this key server. All this information, grouped in different tables, allow to correctly evolve the state of the server among different requests.

Database in key server is also used to log events that occurs inside it. These events are logged in a dedicated table, reporting a message, the level of log and the timestamp related to the moment at which the event was logged. Three log levels have been defined in order to distinguish messages for categories. These level are:

- *INFO*, that refers to any generic event inside the server.
- *WARNING*, that refers to events that may need attention, but do not affect the server functionalities.
- *ERROR*, that refers to error events occurred inside the server.

An high level application can change the current level of events that will be logged as well as access log information by calling dedicated functions. In general, if level is set to *INFO*, all events inside the server will be logged. If level is set to *WARNING*, only error and warning events will be logged. Instead, if level is set to *ERROR*, no events except for the errors will be logged.

As for secure storage, key server is in charge of allowing database access to the underlying QKDMs. Also in this case, during registering phase, key server provides all references needed to QKDM to access its dedicated memory region. In this way a unique database is capable of serving the different QKDMs and the key server itself.

### **Core component**

Core component refers to internal orchestration of resources and information to run the key server. This component is in charge of initialising required resources and managing storage space among different QKDMs.

It may be worth to notice that, in order to manage different resources, key server should already be initialised with a set of information among which the references to storage and database, the list of key server peers it can reach as well as the list of high level applications it serves. At startup, key server may also not be aware of destinations attached to other key servers. When a key exchange is requested with an unknown destination, key server queries all other server peers it finds in its list in order to collect information about the server it needs to contact to reach a specific destination.

When exchanging keys it is possible that some high level application requests particular features, such as employing a specific protocol or considering a given QBER threshold before discarding or keeping a key. Thus, core component must have access to the low level protocols specifications in order to be able to satisfy specific requirements. This information is given by underlying QKDMs during registration phase.

Part of core component regards the communication with other key servers. This is realised through a web interface that will be explained in the next section. The information key servers exchange mainly regards handle of keys to be exchanged or reserved for particular destinations and information related to destinations reachable from particular servers. No sensitive data is sent through this interface, thus a simple public channel can be used. Even if the channel is public, it is however needed to authenticate peer key servers in order to make sure about information origin. Key servers authentication is managed internally by exploiting the quantum exchanged keys coming from underlying level. Basically, when an exchange starts, more than requested key length is exchanged with the destination. The exceeding part of the key is stored to be used for authentication purpose. Authentication between key servers is made by sending a challenge to the server requesting information. The challenge is represented by a string containing the current time, so that it always changes among different requests. The server that receives the challenge must reply by generating an *Hash-based Message Authentication Code* (HMAC) by using one of the keys previously exchanged with the peer server. Hence, it replies by sending the generated HMAC together with the received message and the key handle of the key used during HMAC generation. Key server checks that the sent key handle is actually one of those exchanged with the requesting peer and proceed by verifying the received HMAC. If authentication succeeds peer key server can access the communication interface.

Core component is initialised through a *yaml* configuration file. This file allows to select basic settings such as the preferred QKD protocol for key exchange and the length of keys that will be exchanged. It also used to define the starting log level of the server such as all information needed to correctly talk to database and vault components. Only few of these settings can be changed from the northbound interface when the server is running. If other settings need to be changed, a server manager should act directly on this configuration file by physically access the server.

## 10.2 Interfaces

Key server expose three different interfaces:

- Northbound interface: this is the interface used by high level security applications to talk to key server.
- Southbound interface: interface between key server and underlying QKDMs.
- Internal web interface: represents the public channel used to talk to other key servers.

### Northbound Interface

Northbound interface is used by high level security application to access services key server offers. It basically consists of REST-based methods defined in ETSI standard as shown by table 4.3. These methods allow a basic access of the server functionalities in order to request keys. In a real usage scenario these functions are quite limiting, mainly because they focus on key exchange only and do not allow any other interaction. In order to extend key server functionalities northbound interface integrates additional functions which allow an improved user experience in order to facilitate key server adoption in softwarised networks. An overview of northbound interface can be seen in table 10.1. Its methods will now be described. Note that all data exchanged through this interface are organised in JSON format.

Northbound interface		
Method Name	URL	Access Method
getStatus	https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/status	GET
getKey	https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/enc_keys	POST
getKeyWithKeyIDs	https://{KME_hostname}/api/v1/keys/{master_SAE_ID}/dec_keys	POST
getPreferences	https://{KME_hostname}/api/v1/preferences	GET
setPreference	https://{KME_hostname}/api/v1/preferences/preference	POST
getInfo	https://{KME_hostname}/api/v1/information/info	GET

Table 10.1.

**getStatus** is one of the methods defined by ETSI standard. It is used to retrieve specific destination information related to the keys already available for the requested destination. Basically a high level application (*Secure Application Entity* - SAE) can use this function to know if there are already key available for the destination it wants to exchange the key with and the size of these keys if any. Hence this function is intended to be used before requesting keys. Key server replies to this method by picking desired information from its database.

**getKey** is the method used to retrieve a key (or a set of keys) from the key server. In the request it is possible to specify the number of keys requested together with their size. When keys are requested, key server checks in its storage for available keys and, if it finds keys that meet requirements, it communicates to destination key server the handles of the keys to be reserved for

the SAEs involved in the request. Once keys are reserved, key server knows the peer key server will return the same keys to peer SAE, hence it can safely return the keys to the calling SAE together with their handles that SAE will forward to its peer to let it retrieve the same keys.

**getKeyWithKeyIDs** is the last of the three methods defined by ETSI standard. It is used by SAE that received key handles from peer SAE. Key server that receives this request, checks that requested keys have actually been reserved for this SAE and if all checks went fine, it returns the requested keys.

**getPreferences** method is used to retrieve all the internal settings of the key server that can be changed from SAE. It returns the following kind of information:

- *Log level*: the current level of information that will be logged inside the server. Possible values are INFO, WARNING and ERROR.
- QKD protocol: defines the preferred QKD protocol used when exchanging keys.
- Timeout: it defines the timeout value used when exchanging keys. If a key is not exchanged in the selected timeout value, the exchange is interrupted.

**setPreference** allow to change one of the parameters returned by **getPreferences**. The setting to be changed is specified in the URI of the request, while its value is inside the request body. It is possible to change just one parameter per request hence, if multiple parameters need to be changed, more requests must be issued. If protocol settings are changed with a protocol not managed by attached QKDMs, the request will fail.

**getInfo** is the method used to access information status of the key server. With this method it is possible to retrieve information about the number of attached QKDMs and the protocols they can use. It is also possible using this function to access logged information inside the server. When log is requested, the method allows to filter returned information based on a specific log level or starting by a specific time (or also both of them).

## Southbound Interface

Southbound interface is the interface aimed to communicate with underlying QKD modules. Basically key server uses module interface of QKDM in order to interact to it. However a basic interface has been defined mainly in order to attach new modules. This interface is in fact composed just by one function:

```
def QKD_REGISTER_MODULE(qkdModule, protocol, address)
```

A QKD module that is attached to the server, should call this function by passing the reference to its object as first argument, the protocol it is capable of implement and the address of its web interface. Key server then uses object reference in order to use the module interface of the QKDM itself.

Protocol information is used to select among different QKD modules if a preferred QKD protocol is selected for key exchange.

Address of web interface is used when a new key exchange is started. This information is forwarded to key server peer so that it can instruct its QKDM about the destination it has to communicate with.

Within this function a part of storage is assigned to registered module and its references are returned to it. Besides, key server also returns the maximum number of keys the QKDM can exchange with its peer depending on the memory assigned to it.

Internal web interface		
Method Name	URL	Access Method
challenge	http://{KME_hostname}/api/v1/challenges/kme	GET
authenticate	http://{KME_hostname}/api/v1/authenticate/kme	POST
knownSAE	https://{KME_hostname}/api/v1/saes/slave_SAE_ID	GET
reserveKeys	https://{KME_hostname}/api/v1/kids/master_SAE_ID	POST
openRequest	https://{KME_hostname}/api/v1/keys/	POST
syncRequest	https://{KME_hostname}/api/v1/keys/key	POST

Table 10.2.

### Internal Web Interface

This interface is used by key server to communicate with its peers. As northbound interface it is realised through a web interface and its methods are summarised by table 10.2.

**challenge** is the method used to start authentication. It returns a challenge message that the caller key server must authenticate in order to get access to the other methods of this interface.

**authenticate** method completes the authentication phase. Key server peer uses this method to reply the authentication challenge by sending the HMAC generated with a quantum exchanged key. Hence, if a key server peer calls one of the functions of this interface and get a status code of 401 (Unauthorized), it should run authentication by requesting a challenge with the above method and returning the related HMAC with this one.

**knownSAE** method is used by a key server to query for a given destination that is not initially known. A key server that receives this request, checks if the requested SAE is one of those it serves and, in this case, replies with 200 status code to let requesting server know that it can exchange key with this server when one of its SAE requests a key with the requested SAE as destination. If, instead, requested SAE is not one of those served by this server, it returns 404 status code and the requesting key server will continue to query other key servers in order to find this destination.

**reserveKeys** method is used to reserve keys on this key server for a specific SAE. Reserving keys allows to ensure that key server do not send selected keys to other SAEs before the target SAE receives key handles from the peer SAE and requests them to its key server.

**openRequest** method is used to forward Key\_stream\_ID from a key server to another one as part of key exchange between two QKDMs. This operation is foreseen by QKDM communication protocol as can be seen from section 9.4, where key servers represents *APPA* and *APPB*.

**syncRequest** is used to synchronise index inside the key stream. When requesting a key to QKDM, it is needed to point out the related stream ID and an index referring to a specific key in the stream. This method is used to synchronise requests in order to make sure same indexes are requested to QKDM between two key servers.

Please note that, as for the underlying modules, web interfaces (both the internal and the northbound) run using python Flask module.

## 10.3 Criticalities

Besides the possible issues that may outcome from this implementation, the first criticism that may be worthy to be discussed regards the standard itself. ETSI QKD 014 standard (the one implemented in this chapter) has been though to be built on top of ETSI QKD 004 standard (whose implementation has been discussed in chapter 9). Hence, a stacked implementation composed of QKD Simulator, QKD Module and QKD Key Server has been proposed for this work. However, QKD Module can also be used as a standalone module directly connected to high level applications. Standard states both of these use cases. However in the case in which QKD Key Server is used

on top of QKDM (e.g. this implementation) it does not define a proper interaction between the two layers and some of the concepts in both standards overlap.

While implementing this solution, some of the outlined issues were solved by extending communication interface between QKDM and key server through the southbound interface. Specifically, during registration phase of QKDM, all additional information that standard does not cover are exchanged between the two layers. An example can be seen considering the secure storage in which keys will be saved. Standard does not well define if keys needs to be stored in key server or QKDM, instead it considers a secure storage available in both the modules. When key server has to be used on top of QKDM, keys can be requested to QKDM directly and there is no reason to use two different keys storage. In this implementation key server is a manager of the storage and it provides access to the different QKDM. Besides this specific case, some other little discrepancies emerged during standards study whose lead to the implementation here presented.

A specific criticality of this implementation regards the access to the services it leans on. Basically, authentication server, valut and mysql are in different docker container, hence access to these resources is made through http requests. In implementation phase no https communication has been established. However, when this implementation needs to be deployed in a production environment, these components should be provided of an SSL certificate in order to be accessed through https as to ensure a secure communication.

## 10.4 Workflow

In order to execute a key server it is needed to correctly configure and run all the modules it leans on. Hence it is needed to set up services it uses such as Vault and Keycloak and once everything is ready it is possible to start the server that will raise its web interfaces and will start its core component.

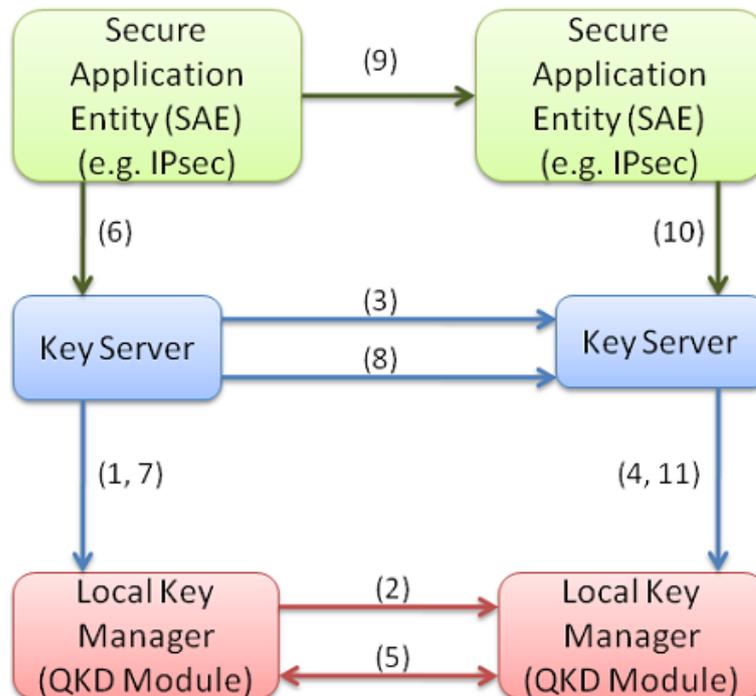


Figure 10.3. Key Server architectural workflow.

An example of workflow involving the proposed architecture is depicted in figure 10.3. Key server is the interface between QKD module that actually exchange keys and high level secure applications. As shown by figure 10.3, key server manages low level modules by initialising them

in order to start keys exchange. These operations are represented by numbers 1 to 4 in the image. Then high level applications access key server in order to retrieve desired keys. Referring to figure 10.3 the architectural workflow is represented by the following steps:

1. Key server calls OPEN\_CONNECT function to its QKD Module.
2. QKD Module register the stream ID generated in OPEN\_CONNECT function to the pair QKDM.
3. Key server register the same stream ID returned to call OPEN\_CONNECT to the peer key server.
4. Peer key server calls OPEN\_CONNECT function as well on its module, completing the synchronisation phase.
5. The two QKDM start to continuously exchange keys.
6. A Secure Application Entity (SAE) requests a key to its key server.
7. Key server requests the key to its QKDM.
8. Before returning the key, key server register to its peer the key ID and reserve that key inside key server peer for the two SAEs involved in the communication.
9. SAE receives key and its ID from key server and forwards the key ID to the peer SAE.
10. Destination SAE requests the same key to its key server by specifying the key ID.
11. Key server retrieves the correct key to its QKDM and returns it to the calling SAE.

It is possible to define functional workflow, by considering all internal components involved in the key server. In this case it is possible to consider two secure application entities called SAEA and SAEB that want to exchange a key by using their key servers called *Key Manager Entity A* (KMEA) and *Key Manager Entity B* (KMEB) respectively, workflow can be summarised by the following steps:

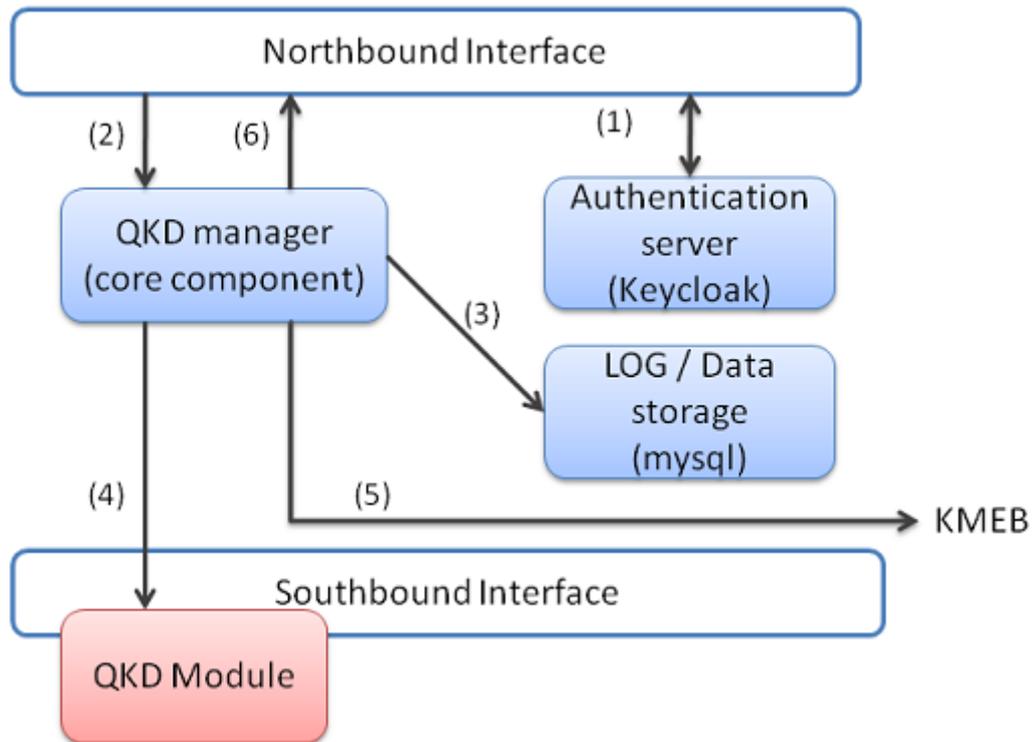


Figure 10.4. getKey method detail.

- KMEA finds KMEB in its key server list and start exchanging keys with it by using functions exposed by QKDM, as described by section 9.4.
- SAEA authenticate itself against Keycloak authentication server and receives a token to be used during communication with its key server.
- Also SAEB authenticates to Keycloak server and obtain an authorisation token.
- SAEA checks if there already are key available for destination SAEB by calling method `getStatus` on northbound interface of KMEA.
- If there are keys available, SAEA requests a key by calling `getKey` status. This method performs a set of steps depicted by figure 10.4 and summarised as follows:
  1. KMEA receives the request and check if access token used as authentication is valid by verifying it through Keycloak.
  2. If token has expired or it is not valid the method returns status code 401 (Unauthorized), otherwise request is forwarded to the core component.
  3. After authentication has been verified, request is logged and KMEA checks if SAEB destination is already known in its database. If not, it fetches the list of KMEs in the network and polls if some of them serves this destination by using `knownSAE` method. After this step KMEA knows the identity of KMEB.
  4. KMEA tries to retrieve desired number of keys from its QKDM. If there are not enough keys KMEA returns an error with a message specifying the error.
  5. KMEA tries to reserve selected key on KMEB by specifying that the key must be reserved for SAEB when it requests that key handle with SAEA as destination. KMEB receive the stream ID and the indexes referring to the keys from KMEA and saves them in the list of reserved keys. Then it returns a status code of 200.
  6. If key reserving is fine, KMEA returns the key together with its ID to the calling SAE.

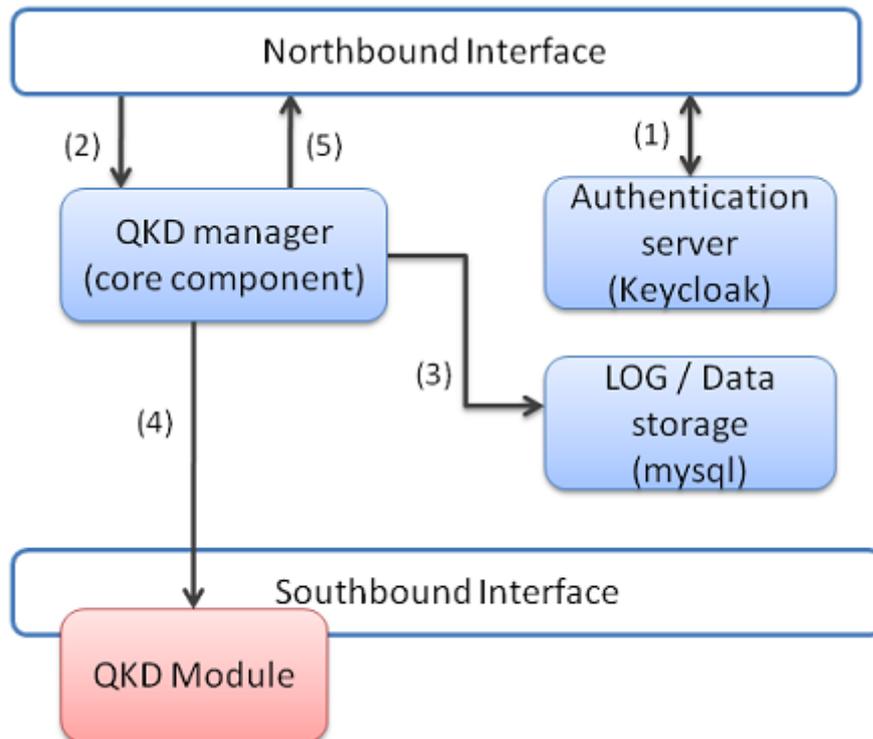


Figure 10.5. `getKeyWithKeyIDs` method detail.

- SAEA receives the key together with its ID from KMEA. It proceeds by forwarding key ID to SAEB so that it can retrieve the same key. Please note that the method SAEA uses to forward key ID to SAEB is out of the scope of this implementation.
- SAEB receives key ID from SAEA and uses this information to retrieve the key by using `getKeyWithKeyIDs` method. This method is depicted by figure 10.5 and its steps can be summarised as follows:
  1. KMEB receives the request and check if access token used as authentication is valid by verifying it through Keycloak.
  2. If token has expired or it is not valid the method returns status code 401 (Unauthorized), otherwise request is forwarded to the core component.
  3. After authentication has been verified, KMEB checks if requested key ID is in the list of reserved keys and if, in this list, the involved KMEs match with KMEB and KMEA specified by the request. If some of these checks fail, KMEB returns an error code together with a message specifying the error.
  4. If checks are fine, KMEB retrieves key associated to requested key ID from the underlying QKDM module and deletes the key ID from the list of reserved keys.
  5. KMEB returns the key to the calling SAE.
- Now, SAEA and SAEB share the same secret that can be used for cryptographic purposes.

After this overview, a final consideration can be done on the communication channels involved in the proposed architecture. A simple schema highlighting communication channels can be seen in figure 10.6. QKD system in figure corresponds to the proposed architecture composed of QKD

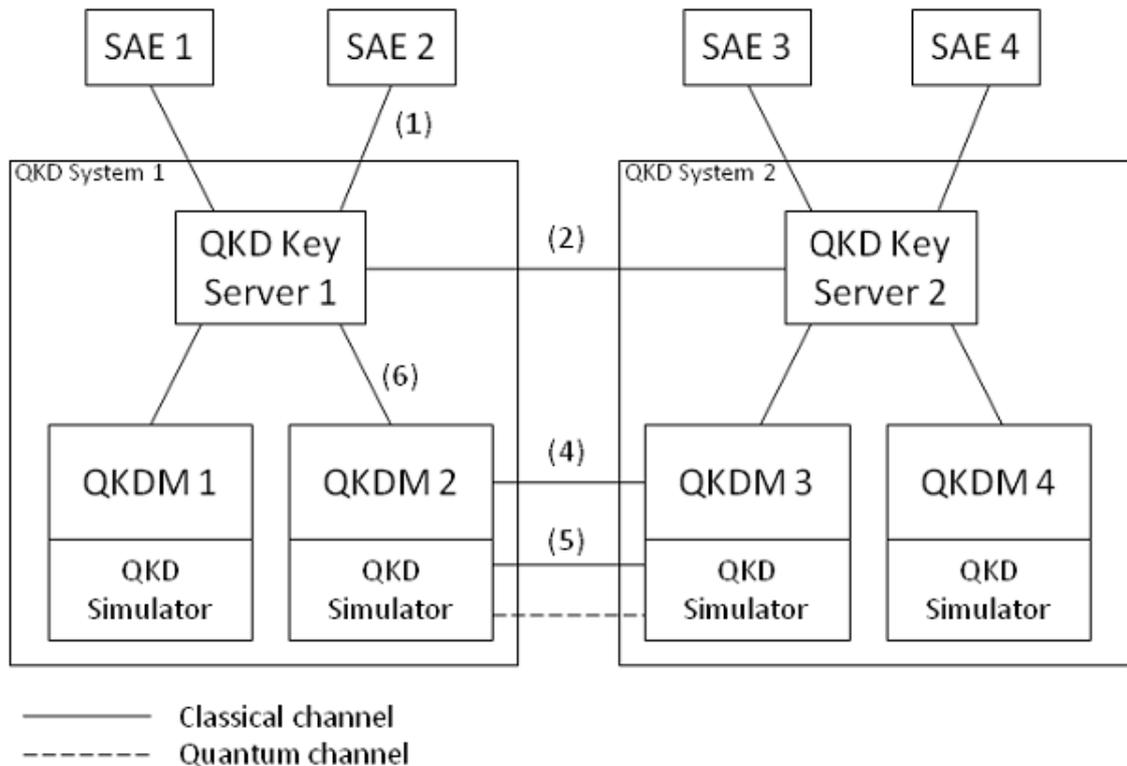


Figure 10.6. Final architecture with its communications channel.

key server, QKD module and QKD simulator. As already seen, this system must be able to communicate horizontally with its pairs and vertically with its internal components and external SAEs. A description of the communication channels involved has been given during the previous

chapters while describing the single components, however it may be worthy to summarise these considerations looking at the whole architecture.

Horizontal communication channels (numbers 2, 4 and 5 in figure 10.6) are subject to the same considerations. Information exchanged on these channels can be accessed by an adversary without invalidate the overall security since, even if the information are related to the key, they do not leak any data on the key itself. For these channels is however crucial to be able to identify the source of the exchanged data, to make sure no one modified data inside the channel. Hence these channels need to be authenticated. Data authentication can be achieved through different strategy. As already said when discussed QKD simulator, here post quantum cryptography and preshared keys methods were investigated. Post quantum cryptography allows to use asymmetric encryption, requiring a lower number of keys to be managed. However, even if preshared keys requires a higher number of keys, the proposed architecture is capable of generating them when needed, making preshared keys a valid solution.

Vertical communication can be divided into two categories: communication inside the QKD system itself and communication between the QKD system and a high level application (SAE). Communication inside the QKD system is represented by number 6 of figure 10.6. This channel transmits sensitive information (e.g. the exchanged keys) and must then be secured. In simpler architectures key server and QKD module resides on the same machine. In this case it is enough to guarantee offline security with secure memory regions. If QKD module resides on a machine different from key server, the whole communication needs to be secured, by means of confidentiality, authenticity and integrity. If such distributed architecture need to be set up, further consideration on how to achieve such security should be done.

Communication between key server and SAE (number 1 in figure 10.6) occurs through REST API. Since key server communicate keys to the SAE, this channel needs to be secured against any kind of interference. In this case, besides data encryption, SAE should be able to authenticate key server as well as key server should be able to authenticate SAE to make sure only authorised parties can access data. As already said, authentication of SAE is managed through keycloak. Authentication of key server, as well as data encryption is currently realised by communicating with https protocol. In this case servers have been provided of a TLS certificate, signed by a self generated certification authority. However, this solution is not scalable and further study may identify a more proper security method for this channel.

# Chapter 11

## Test and validation

The proposed solution has been widely tested in order to understand its limitations and to identify possible improvements. Two different system has been tested: the QKD Simulator presented in chapter 8 and the system composed of QKD Key Server and QKD Module which leans on a simpler version of QKD Simulator.

In order to simulate a distributed environment, each instance involved in communication should be executed on a different machine. To test this environment docker containers have been used. Containers orchestration has been managed through docker compose, in order to manage them all with a single configuration file.

All tests were run on a virtual machine running Ubuntu 20.10 LTS with 3GB of RAM, 40GB of Solid State Drive. The host system is a Dell XPS with Windows 10 and a CPU Intel Core i7-7700HQ. Employed software components use the following versions:

- Docker version 19.03.13.
- Docker-compose version 1.27.4.
- Mysql docker version 8.0.22.
- Vault docker version 1.6.0.
- Keycloak docker version 11.0.3.
- Python version 3.7.9 with the following modules:
  - Qiskit version 0.22.0.
  - Flask version 1.1.1.
  - Mysqlq-connector version 8.0.20.
  - Hvac version 0.10.4.
  - Requests version 2.23.0.
  - Cryptography version 2.8.
  - Pyspx version 0.4.0.

### 11.1 QKD Simulator

To test QKD Simulator a dedicated environment has been set up. A different docker container has been set up for each of involved components. Docker compose file sets up basic configuration and organizes all container in the same network so that they can reach each other. Referring to figure 8.1 docker compose file has been written to set up the following containers:

- A QKD Simulator instance representing source with address 172.15.0.4.
- A QKD Simulator instance representing detector with address 172.15.0.5.
- A channel instance (both classical and quantum) with address 172.15.0.6.
- A container for generating singlets for E91 protocol with address 172.15.0.7.
- A container to manage the whole system with address 172.15.0.8.
- A mysql server with address 172.15.0.2.
- A vault server with address 172.15.0.3.

Test was run by accessing the simulator manager and launching key exchanges with different parameters. The first test was aimed to identify the best statevector length to minimise exchange time. Considering BB84 protocol, *SPHINCS+* authentication method and a key length of 128 bits, statevector length causes a variation of exchanging time as depicted by figure 11.1.

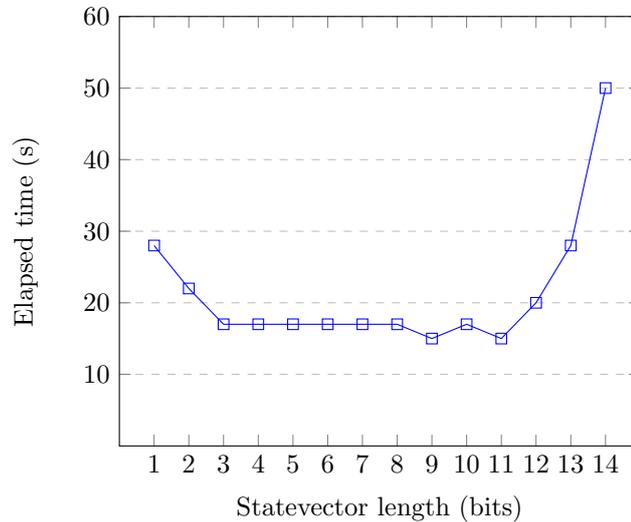


Figure 11.1. Key exchange required time for different statevector lengths (BB84, *SPHINCS+* and 128 bits key).

The figure shows an average time of 17 seconds for the key exchange if the statevector length is between 3 and 11 bits. For short length like 1 or 2 performance decreases up to about 30 seconds. The worst case is for a length of 14 bits, in which a key exchange can last up to 50 seconds. Note that simulate a key exchange with a statevector length of 15 bits or beyond raises memory error, this is why tests stop at statevector length of 14 bits. As can be seen from figure, the best case in the range 1 - 14 is given by the case in which statevector length is 9. In this case key is exchanged within 15 seconds. For BB84 protocol and *SPHINCS+* authentication method a statevector length of 9 bits has been used for next tests.

Once the best statevector length has been detected, keys with different length were exchanged in order to identify the bit rate of the system. Multiple run of the same test were executed in order to identify the mean time required by each transfer. The results can be seen in figure 11.2.

Image shows a quite linear trend of the required time: every time the key length is doubled, also the required time doubles. For a 128 bits key required time is 15 seconds. It becomes 23 seconds for a 256 bits key and 43 seconds for a 512 bits key. From here, doubling the number of bits results in doubling required time: for 1024 bits 86 seconds are required, 168 for 2048 bits, 332 for 4096 bits and so forth. The system hence is capable of exchanging keys at a bit rate around 12 bits per second.

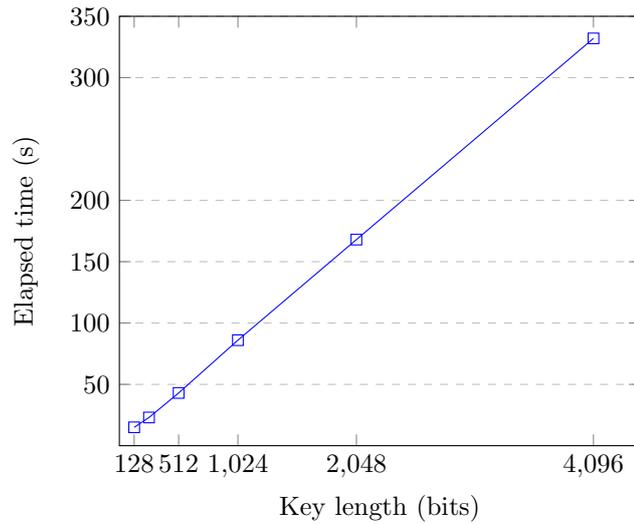


Figure 11.2. Key exchange required time for different key lengths (BB84, *SPHINCS*<sup>+</sup> and statevector length of 9 bits).

The same tests were run with AES-GCM authentication method and results are comparable to the case of *SPHINCS*<sup>+</sup>. A graph showing time variation at different key length with AES-GCM authentication can be seen in figure 11.3.

Considering that, from tests carried out, AES-GCM does not introduce any variation in terms of time with respect to *SPHINCS*<sup>+</sup> an analysis of the required time by each stage of BB84 protocol has been performed in order to identify improvements points. At key length variation, six different stages were analysed:

1. Qubits generation time at source ( $t_q$ ).
2. `sendRegister` method time ( $t_s$ ).
3. *SPHINCS*<sup>+</sup> signature / AES-GCM encryption ( $t_{auth}$ ).
4. `compareBasis` method time ( $t_c$ ).

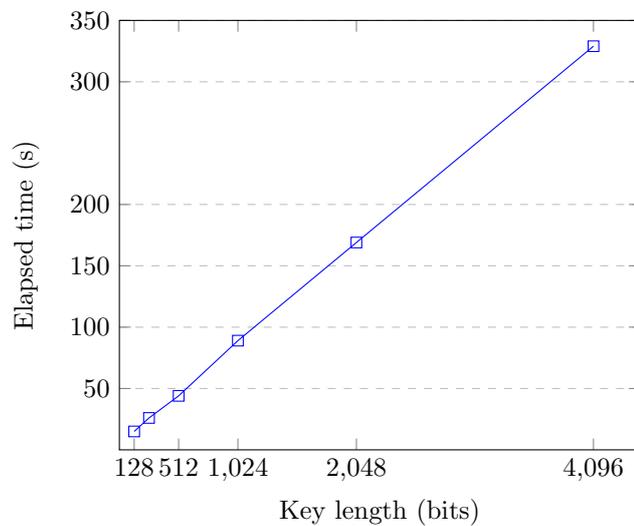


Figure 11.3. Key exchange required time for different key lengths (BB84, AES-GCM and statevector length of 9 bits).

5. *SPHINCS*<sup>+</sup> verify / AES-GCM decryption ( $t_{\text{verify}}$ ).
6. Key sifting time ( $t_{\text{ks}}$ ).

The overall time is given by the sum of the previous stages:

$$t_{\text{exchange}} = t_{\text{q}} + t_{\text{s}} + t_{\text{auth}} + t_{\text{c}} + t_{\text{verify}} + t_{\text{ks}} \quad (11.1)$$

Performed tests show how step 1 has a huge impact on the overall timings. Considering the case of 128 bits key, qubits generation requires 7 seconds over 15 required by the whole exchange. Increasing key length cause this time to grow consequently. Considering the 332 seconds employed for the exchange of 4096 bits key, 220 seconds are required by qubits generation. This step requires more than the half of the overall time and future works may focus on improving this phase to optimize the system. The other time consuming step is key sifting. In this phase the whole basis table must be parsed in order to identify bits of the key that need to be discarded. Thus, time required by this operation is strictly dependant by the length of the key and represents about the 20% of the overall time. Even if in lower measure, even step 2 affects system performances. This is due to the fact that inside `sendRegister` method, simulator instance performs the measurements of received qubits. This time varies from 1 seconds for 128 bits key to 32 seconds for 4096 bits key, showing a minor impact on the overall timings. Steps 3, 4 and 5 are negligible times compared to the others. `compareBasis` method simply forwards basis table hence, time required by this step is reduced to the network time needed to send data. The maximum measured time for this step is 0.5 second for 4096 bits key. *SPHINCS*<sup>+</sup> and AES-GCM operations require even less time than `compareBasis` method. In this case it is possible to see the different timings required by these two algorithms. *SPHINCS*<sup>+</sup> signature is not affected by the increasing length of the data to be sign and it requires about 0.5 seconds. Signature verifying is even faster, requiring about 0.05 seconds. AES-GCM is capable of performing its operations with an order of magnitude less than *SPHINCS*<sup>+</sup>. Encryption time is about 0.07 seconds, whist decryption only require 0.005 seconds. However, the small contribution given from authentication phase does not let appreciate any difference between the two when the overall time is considered.

To test system reliability an attack scenario was simulated. Test was run with Intercept and Resend attack over BB84, in which an eavesdropper (modelled inside the channel), measures qubits sent before forwarding them to the destination. As can be seen from figure 11.4, from the result of the key exchange emerges that the two keys are different. This result comes out from the verification phase in which compared bits were different. In this case the key is returned to the calling application together with a *False* flag, stating that verification failed. If eavesdropper measures just a portion of exchanged qubits the error rate is decreased. This linear trend is due to the ideal condition of this simulation in which channel does not introduce error and every error is addressable to eavesdropper.

One last test regarding simulator was run with E91 protocol. Even if E91 protocol cannot be correctly implemented in a simulation as already discussed in chapter 8, a test of the proposed solution were run in order to individuate the maximum key rate achievable with this customised version of the protocol. Even in this case, multiple key exchanges with different lengths were requested and results can be seen in figure 11.5. Figure shows the same trend already seen for BB84 protocol. By doubling the length of the key the required time is doubled. The interesting data is that the proposed implementation requires more than the double of the time required from BB84 protocol for the same key length. For a 128 bits key required time is 28 seconds that is comparable to the time required for a 256 bits keys with BB84. 256 bits key require 56 seconds and, for 512, 1024, 2048 and 4096 bits key, required time is 112, 233, 444 and 984 seconds respectively.

## 11.2 QKD Key Server and Module

The entire system composed of QKD Key Server, QKD Module and a simplified version of QKD Simulator has been deployed in a different environment from the one specified in the previous section. A dedicated docker compose has been written, in order to setup all needed containers in the same subnetwork. The current configuration involves the following containers:

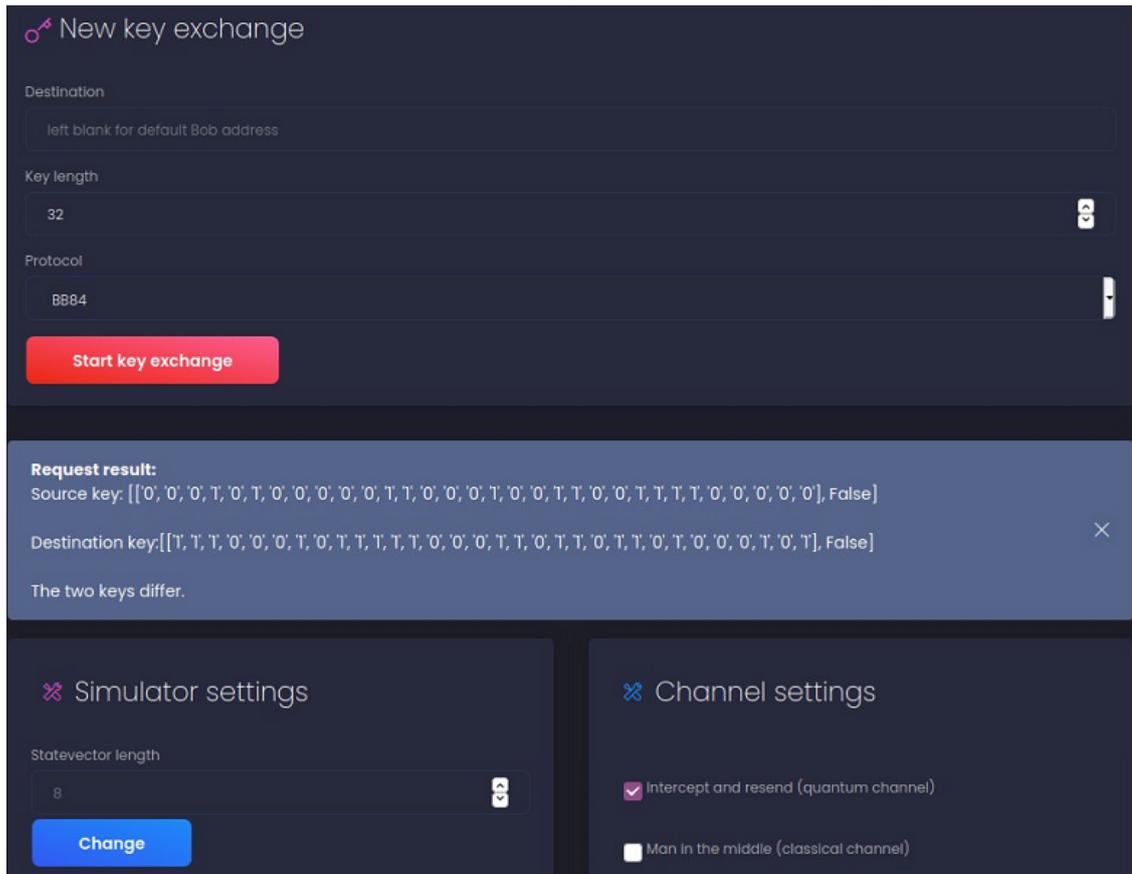


Figure 11.4. Key exchange with Intercept and Resend attack.

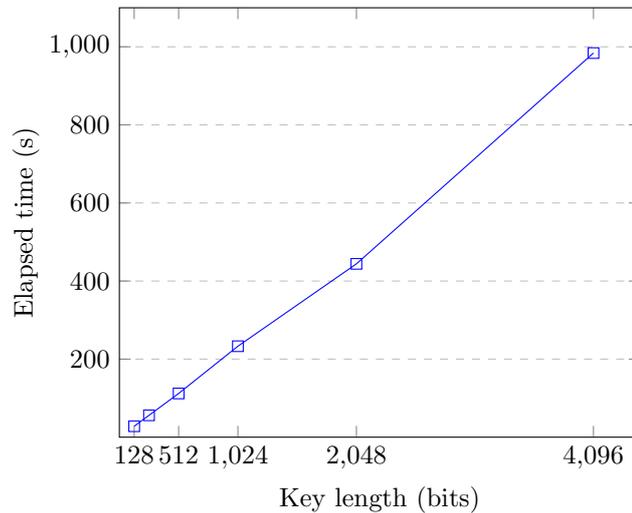


Figure 11.5. Key exchange required time for different key lengths (E91 and statevector length of 9 bits).

- A container running QKD Key server instance, attached QKD Modules and their simulator implementation with address 172.17.0.6
- Another instance of the above container representing another QKD Key server in the distributed environment with address 172.17.0.7.

- A mysql server with address 172.17.0.2.
- A keycloak server with address 172.17.0.3.
- A vault server, serving the first QKD key server instance with address 172.17.0.4.
- Another vault server, serving the second QKD key server instance with address 172.17.0.5.

Once all needed containers are running, in order to test the system it is needed to use high level applications that interface with its northbound interface. To this aim a python application was developed, in order to interface to the two different key servers running in related containers to request keys. This test application should be run twice simultaneously, in order to connect to the two available key servers. Note that high level applications implementation is out of the scope of this work. Here a simple python script has been written with the only aim of requesting operations to key server in order to test its functionalities.

Test script provides a simple command line interface to requests for operations. This interface can be seen in figure 11.6. Comparing this interface to table 10.1, it can be easily seen how each

```
Please insert command:
1) Get status
2) Get key
3) Get key with key ID
4) Get preferences
5) Set preference
6) Get info
7) Exit
[1-7]: 
```

Figure 11.6. Test script command line interface.

one of the entries the interface provides matches one of the methods of the northbound interface. In particular, the entries from 1 to 6 correspond to the following methods:

1. `getStatus()`
2. `getKey()`
3. `getKeyWithKeyIDs()`
4. `getPreferences()`
5. `getInfo()`

When one of this entry is selected, script connects to the selected key server and performs required operation. Key server address, as well as other specific application configurations are hardcoded in the script itself.

In order to operate on the selected key server, test script needs to be authenticated. Hence, before the test starts, it is needed to retrieve an authentication token from keycloak server. To get an access token a client needs to be registered on keycloak. After registration, client will get an ID and a secret which can be used to request tokens. Registration of application client is done manually through the keycloak web interface (more details regarding client registration will be provided in appendix A. If registration phase can be done once, every time the test is started a new token must be requested. This because validity of tokens expires after a time that is selected during client registration. Tokens are required through POST requests to the specific realm endpoint. The body of the request should include the client ID and its secret and the header `Content-type` must be mandatory set to `application/x-www-form-urlencoded`. An example requests of token that uses python language is the following:



## Chapter 12

# Conclusions

The proposed work tried to provide a new approach for Quantum Key Exchange. Although the research is moving forward to identify technologies and protocols to improve QKD, few studies focus their attention on integrate such systems in existing environments such as distributed networks. ETSI standard moves in this direction, defining a simple interface to key server accessible to REST API. However, this work outlined some criticalities and proposed some improvements to the overall system. Proposed implementation was written in a high level language in order to be easily accessible for modification. Despite some implementation flaws, the result is a robust architecture that can be easily adopted and modified as needed.

Besides key server, this implementation propose a different implementation for QKD simulator. The need of a separate development for the QKD simulator was not foreseen in the initial phase. This need came out during the development itself and led to the implementation proposed in chapter 8. This specific work was proven to be useful, not for the key server implementation itself, but for other researches related to QKD, since current implementations of QKD simulators are based on exchange within the same machine, with few attention to third party attacks or channel modelling.

Tests performed on QKD simulator show a flexible architecture with some criticalities on the time needed for a single key exchange. Optimisations on QKD algorithms implementation can improve required timings. Besides the employ of density matrices rather than statevector can be investigated to further reduce this metric. Other developments for this module may be represented by the addition of mathematical models to represent a non-ideal scenarios (with decoherence and errors on quantum channel) that were not investigated in this stage.

Regarding the overall architecture, performed tests show the impact of a QKD Key server in a distributed environment. The proposed architecture is capable of serving different security applications, lying on more QKD modules capable of sharing keys exploiting different protocols. The results demonstrate the low impact of the high level modules with respect to the time required by a quantum key exchange. This difference can be seen by looking at key rate reached when a dummy simulator is used (with keys are immediately available at each requests) and when a real BB84 implementation is used.

Future work on this subject may regard the employment of real QKD devices to check the flexibility of the proposed implementation when devices coming from different producers are used. Instead, a more direct evolution to this work regards the usage of the key server in a real distributed environment such as OpenStack in order to identify improvements that may be needed in a real distributed environment.

# Bibliography

- [1] A. G. White, D. F. V. James, W. J. Munro, and P. G. Kwiat, “Exploring Hilbert space: Accurate characterization of quantum information”, *Physical Review A*, vol. 65, December 2001, pp. 1–4, DOI [10.1103/physreva.65.012301](https://doi.org/10.1103/physreva.65.012301)
- [2] J. K. Vizzotto, “Quantum computing: State-of-art and challenges”, 2nd Workshop-School on Theoretical Computer Science, Rio Grande (Brazil), October 15-17, 2013, pp. 9–13, DOI [10.1109/WEIT.2013.34](https://doi.org/10.1109/WEIT.2013.34)
- [3] A. Pasiëka, D. Kribs, R. Laflamme, and R. Pereira, “On the Geometric Interpretation of Single Qubit Quantum Operations on the Bloch Sphere”, *Acta Appl Math*, vol. 108, September 2009, pp. 697–707, DOI [10.1007/s10440-008-9423-z](https://doi.org/10.1007/s10440-008-9423-z)
- [4] P. A. M. Dirac, “A new notation for quantum mechanics”, *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35, July 1939, pp. 416–418, DOI [10.1017/S0305004100021162](https://doi.org/10.1017/S0305004100021162)
- [5] G. L. Khym and H. Yang, “Quantum entanglement does not violate the principle of special theory of relativity”, *Physics Essays*, vol. 29, December 2016, pp. 553–554, DOI [10.4006/0836-1398-29.4.553](https://doi.org/10.4006/0836-1398-29.4.553)
- [6] J. S. Bell, “On the Einstein Podolsky Rosen paradox”, *Physics Physique Fizika*, vol. 1, November 1964, pp. 195–200, DOI [10.1103/PhysicsPhysiqueFizika.1.195](https://doi.org/10.1103/PhysicsPhysiqueFizika.1.195)
- [7] M. Born, “Quantum mechanics of collision processes”, *Zeit. Physik*, vol. 37, June 1926, pp. 863–867, DOI [10.1007/BF01397477](https://doi.org/10.1007/BF01397477). [http://www.neo-classical-physics.info/uploads/3/0/6/5/3065888/born\\_-\\_qm\\_for\\_collisions\\_ii.pdf](http://www.neo-classical-physics.info/uploads/3/0/6/5/3065888/born_-_qm_for_collisions_ii.pdf)
- [8] H. E. Brandt, “Positive-operator and projection valued measurements in quantum key distribution”, *Journal of Modern Optics*, vol. 54, August 2007, pp. 2357–2363, DOI [10.1080/09500340701639557](https://doi.org/10.1080/09500340701639557)
- [9] C. E. Shannon, “Communication Theory of Secrecy Systems”, *Bell System Technical Journal*, vol. 28, October 1949, pp. 656–715, DOI [10.1002/j.1538-7305.1949.tb00928.x](https://doi.org/10.1002/j.1538-7305.1949.tb00928.x)
- [10] P. W. Shor, “Algorithms for quantum computation: discrete logarithms and factoring”, 35th Annual Symposium on Foundations of Computer Science, Santa Fe (NM, USA), 1994, pp. 124–134, DOI [10.1109/SFCS.1994.365700](https://doi.org/10.1109/SFCS.1994.365700)
- [11] M. Walker, “Hype Cycle for Emerging Technologies”, Gartner, August 2018. <https://www.gartner.com/en/documents/3885468/hype-cycle-for-emerging-technologies-2018>
- [12] C. Gidney and M. Ekerå, “How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits”, arXiv, Dec 2019, pp. 1–26. <https://arxiv.org/abs/1905.09749>
- [13] O. Regev, “On Lattices, Learning with Errors, Random Linear Codes, and Cryptography”, *Journal of the ACM*, vol. 56, September 2009, pp. 1–40, DOI [10.1145/1568318.1568324](https://doi.org/10.1145/1568318.1568324)
- [14] K. Basu, D. Soni, M. Nabeel, and R. Karri, “NIST Post-Quantum Cryptography- A Hardware Evaluation Study”, *Cryptology ePrint Archive, Report 2019/047*, vol. 47, May 2019, pp. 1–16. <https://eprint.iacr.org/2019/047>
- [15] B. Biswas and N. Sendrier, “McEliece cryptosystem implementation: Theory and practice”, Springer Berlin Heidelberg, 2008, ISBN: 978-3-540-88402-6
- [16] P. K. S. Rao, D. M. Krishna, and D. Ravi, “Multivariate Public Key Cryptography and Digital Signature”, arXiv, July 2018, pp. 1–30. <https://arxiv.org/abs/1807.07710>
- [17] S. Wiesner, “Conjugate Coding”, *SIGACT News*, vol. 15, January 1983, pp. 78–88, DOI [10.1145/1008908.1008920](https://doi.org/10.1145/1008908.1008920)
- [18] D. Gottesman and I. Chuang, “Quantum Digital Signatures”, arXiv, May 2001, pp. 1–8. <https://arxiv.org/abs/quant-ph/0105032>

- [19] R. J. Collins, R. Amiri, M. Fujiwara, T. Honjo, K. Shimizu, K. Tamaki, M. Takeoka, E. Andersson, G. S. Buller, and M. Sasaki, “Experimental transmission of quantum digital signatures over 90 km of installed optical fiber using a differential phase shift quantum key distribution system”, *Optics Letter*, vol. 41, November 2016, pp. 4883–4886, DOI [10.1364/OL.41.004883](https://doi.org/10.1364/OL.41.004883)
- [20] H.-L. Huang, Q. Zhao, X. Ma, C. Liu, Z.-E. Su, X.-L. Wang, L. Li, N.-L. Liu, B. C. Sanders, C.-Y. Lu, and J.-W. Pan, “Experimental Blind Quantum Computing for a Classical Client”, *Phys. Rev. Lett.*, vol. 119, August 2017, pp. 1–8, DOI [10.1103/PhysRevLett.119.050503](https://doi.org/10.1103/PhysRevLett.119.050503)
- [21] D. Mayers, “Unconditionally Secure Quantum Bit Commitment is Impossible”, *Physical Review Letters*, vol. 78, April 1997, pp. 3414–3417, DOI [10.1103/PhysRevLett.78.3414](https://doi.org/10.1103/PhysRevLett.78.3414)
- [22] N. Ng, S. Joshi, C. Chen Ming, C. Kurtsiefer, and S. Wehner, “Experimental implementation of bit commitment in the noisy-storage model”, *Nature Communications*, vol. 3, May 2012, pp. 1–7, DOI [10.1038/ncomms2268](https://doi.org/10.1038/ncomms2268)
- [23] W. Wootters and W. Zurek, “A single quantum cannot be cloned”, *Nature*, vol. 299, October 1982, pp. 802–803, DOI [10.1038/299802a0](https://doi.org/10.1038/299802a0)
- [24] X. Ma, X. Yuan, Z. Cao, B. Qi, and Z. Zhang, “Quantum random number generation”, *npj Quantum Information*, vol. 2, June 2016, pp. 1–9, DOI [10.1038/npjqi.2016.21](https://doi.org/10.1038/npjqi.2016.21)
- [25] C. Portmann and R. Renner, “Cryptographic security of quantum key distribution”, arXiv, September 2014, pp. 1–58. <https://arxiv.org/abs/1409.3525>
- [26] P. Kurpiers, M. Pechal, B. Royer, P. Magnard, T. Walter, J. Heinsoo, Y. Salathé, A. Akin, S. Storz, J.-C. Besse, and et al., “Quantum Communication with Time-Bin Encoded Microwave Photons”, *Physical Review Applied*, vol. 12, October 2019, pp. 1–11, DOI [10.1103/physrevapplied.12.044067](https://doi.org/10.1103/physrevapplied.12.044067)
- [27] J. C. Campbell, B. C. Johnson, G. J. Qua, and W. T. Tsang, “Frequency response of InP/InGaAsP/InGaAs avalanche photodiodes”, *Journal of Lightwave Technology*, vol. 7, May 1989, pp. 778–784, DOI [10.1109/50.19113](https://doi.org/10.1109/50.19113)
- [28] P. W. Shor and J. Preskill, “Simple Proof of Security of the BB84 Quantum Key Distribution Protocol”, *Physical Review Letters*, vol. 85, July 2000, pp. 441–444, DOI [10.1103/physrevlett.85.441](https://doi.org/10.1103/physrevlett.85.441)
- [29] P. W. Shor, “Scheme for reducing decoherence in quantum computer memory”, *Physical Review A*, vol. 52, October 1995, pp. 2493–2496, DOI [10.1103/PhysRevA.52.R2493](https://doi.org/10.1103/PhysRevA.52.R2493)
- [30] M. Toyran, T. B. Pedersen, A. S. Atilla Hasekioglu, M. Ali Can, and S. Berber, “A study on cascade error correction protocol”, 2013 21st Signal Processing and Communications Applications Conference (SIU), Haspolat (Turkey), April 24–26, 2013, pp. 1–4, DOI [10.1109/SIU.2013.6531338](https://doi.org/10.1109/SIU.2013.6531338)
- [31] K. Zigangirov, A. Pusane, D. Zigangirov, and D. Costello, “On the Error-Correcting Capability of LDPC Codes”, *Problems of Information Transmission*, vol. 44, October 2008, pp. 214–225, DOI [10.1134/S0032946008030046](https://doi.org/10.1134/S0032946008030046)
- [32] M. M. Khan, M. Murphy, and A. Beige, “High error-rate quantum key distribution for long-distance communication”, *New Journal of Physics*, vol. 11, June 2009, pp. 1–17, DOI [10.1088/1367-2630/11/6/063043](https://doi.org/10.1088/1367-2630/11/6/063043)
- [33] C. Bennett and G. Brassard, “Quantum cryptography: Public key distribution and coin tossing”, *Theoretical Computer Science - TCS*, vol. 560, December 2014, pp. 7–11, DOI [10.1016/j.tcs.2011.08.039](https://doi.org/10.1016/j.tcs.2011.08.039)
- [34] A. K. Ekert, “Quantum cryptography based on Bell’s theorem”, *Physical review letters*, vol. 67, August 1991, pp. 661–663, DOI [10.1103/PhysRevLett.67.661](https://doi.org/10.1103/PhysRevLett.67.661)
- [35] M. Lucamarini, K. A. Patel, J. F. Dynes, B. Fröhlich, A. W. Sharpe, A. R. Dixon, Z. L. Yuan, R. V. Penty, and A. J. Shields, “Efficient decoy-state quantum key distribution with quantified security”, *Optics express*, vol. 21, October 2013, pp. 24550–24565, DOI [10.1364/OE.21.024550](https://doi.org/10.1364/OE.21.024550)
- [36] H.-K. Lo, X. Ma, and K. Chen, “Decoy State Quantum Key Distribution”, *Physical Review Letters*, vol. 94, June 2005, pp. 1–4, DOI [10.1103/PhysRevLett.94.230504](https://doi.org/10.1103/PhysRevLett.94.230504)
- [37] S. Ali, S. Mohammed, M. Chowdhury, and A. Hasan, “Practical SARG04 quantum key distribution”, *Optical and Quantum Electronics*, vol. 44, March 2012, pp. 471–482, DOI [10.1007/s11082-012-9571-2](https://doi.org/10.1007/s11082-012-9571-2)

- [38] H. Bechmann-Pasquinucci and N. Gisin, “Incoherent and coherent eavesdropping in the six-state protocol of quantum cryptography”, *Physical Review A*, vol. 59, June 1999, pp. 4238–4248, DOI [10.1103/PhysRevA.59.4238](https://doi.org/10.1103/PhysRevA.59.4238)
- [39] C. Bennett, “Quantum cryptography using any two nonorthogonal states”, *Physical review letters*, vol. 68, May 1992, p. 3121, DOI [10.1103/PhysRevLett.68.3121](https://doi.org/10.1103/PhysRevLett.68.3121)
- [40] Bennett, Brassard, and Mermin, “Quantum cryptography without Bell’s theorem”, *Physical review letters*, vol. 68, February 1992, p. 557, DOI [10.1103/PhysRevLett.68.557](https://doi.org/10.1103/PhysRevLett.68.557)
- [41] Y. Mu, J. Seberry, and Y. Zheng, “Shared cryptographic bits via quantized quadrature phase amplitudes of light”, *Optics communications*, vol. 123, January 1996, pp. 344–352, DOI [10.1016/0030-4018\(95\)00688-5](https://doi.org/10.1016/0030-4018(95)00688-5)
- [42] A. I. Khaleel, “Coherent one-way protocol: Design and simulation”, *International Conference on Future Communication Networks*, Baghdad (Iraq), June 1, 2012, pp. 170–174, DOI [10.1109/ICFCN.2012.6206863](https://doi.org/10.1109/ICFCN.2012.6206863)
- [43] K. Inoue, H. Takesue, and T. Honjo, “DPS quantum key distribution and related technologies”, *SPIE - The International Society for Optical Engineering*, vol. 7236, January 2009, pp. 1–9, DOI [10.1117/12.808590](https://doi.org/10.1117/12.808590)
- [44] J. Mower, Z. Zhang, P. Desjardins, C. Lee, J. H. Shapiro, and D. Englund, “High-dimensional quantum key distribution using dispersive optics”, *Physical Review A*, vol. 87, June 2013, pp. 1–8, DOI [10.1103/PhysRevA.87.062322](https://doi.org/10.1103/PhysRevA.87.062322)
- [45] M. Epping, H. Kampermann, and D. Bruß, “Designing Bell inequalities from a Tsirelson bound”, *Physical review letters*, vol. 111, December 2013, pp. 1–5, DOI [10.1103/PhysRevLett.111.240404](https://doi.org/10.1103/PhysRevLett.111.240404)
- [46] U. Vazirani and T. Vidick, “Fully device-independent quantum key distribution”, *Communications of the ACM*, vol. 62, March 2019, pp. 1–10, DOI [10.1145/3310974](https://doi.org/10.1145/3310974)
- [47] European Telecommunications Standards Institute, “Quantum Key Distribution (QKD); Vocabulary”, December 2008, [https://www.etsi.org/deliver/etsi\\_gr/QKD/001\\_099/007/01.01.01\\_60/gr\\_QKD007v010101p.pdf](https://www.etsi.org/deliver/etsi_gr/QKD/001_099/007/01.01.01_60/gr_QKD007v010101p.pdf)
- [48] European Telecommunications Standards Institute, “Quantum Key Distribution (QKD); Device and Communication Channel Parameters for QKD Deployment”, February 2019, [https://www.etsi.org/deliver/etsi\\_gs/QKD/001\\_099/012/01.01.01\\_60/gs\\_QKD012v010101p.pdf](https://www.etsi.org/deliver/etsi_gs/QKD/001_099/012/01.01.01_60/gs_QKD012v010101p.pdf)
- [49] European Telecommunications Standards Institute, “Quantum Key Distribution (QKD); Components and Internal Interfaces”, March 2018, [https://www.etsi.org/deliver/etsi\\_gr/QKD/001\\_099/003/02.01.01\\_60/gr\\_QKD003v020101p.pdf](https://www.etsi.org/deliver/etsi_gr/QKD/001_099/003/02.01.01_60/gr_QKD003v020101p.pdf)
- [50] European Telecommunications Standards Institute, “Quantum Key Distribution (QKD); Component characterization: characterizing optical components for QKD systems”, May 2016, [https://www.etsi.org/deliver/etsi\\_gs/QKD/001\\_099/011/01.01.01\\_60/gs\\_QKD011v010101p.pdf](https://www.etsi.org/deliver/etsi_gs/QKD/001_099/011/01.01.01_60/gs_QKD011v010101p.pdf)
- [51] European Telecommunications Standards Institute, “Quantum Key Distribution (QKD); Security Proofs”, December 2010, [https://www.etsi.org/deliver/etsi\\_gs/QKD/001\\_099/005/01.01.01\\_60/gs\\_QKD005v010101p.pdf](https://www.etsi.org/deliver/etsi_gs/QKD/001_099/005/01.01.01_60/gs_QKD005v010101p.pdf)
- [52] European Telecommunications Standards Institute, “Quantum Key Distribution (QKD); Use Cases”, June 2010, [https://www.etsi.org/deliver/etsi\\_gs/qkd/001\\_099/002/01.01.01\\_60/gs\\_qkd002v010101p.pdf](https://www.etsi.org/deliver/etsi_gs/qkd/001_099/002/01.01.01_60/gs_qkd002v010101p.pdf)
- [53] European Telecommunications Standards Institute, “Quantum Key Distribution (QKD); Application Interface”, August 2020, [https://www.etsi.org/deliver/etsi\\_gs/QKD/001\\_099/004/02.01.01\\_60/gs\\_QKD004v020101p.pdf](https://www.etsi.org/deliver/etsi_gs/QKD/001_099/004/02.01.01_60/gs_QKD004v020101p.pdf)
- [54] European Telecommunications Standards Institute, “Quantum Key Distribution (QKD); Protocol and data format of REST-based key delivery API”, February 2019, [https://www.etsi.org/deliver/etsi\\_gs/QKD/001\\_099/014/01.01.01\\_60/gs\\_QKD014v010101p.pdf](https://www.etsi.org/deliver/etsi_gs/QKD/001_099/014/01.01.01_60/gs_QKD014v010101p.pdf)
- [55] L. Salvail, M. Peev, E. Diamanti, R. Alléaume, N. Lütkenhaus, and T. Länger, “Security of Trusted Repeater Quantum Key Distribution Networks”, *Journal of Computer Security*, vol. 18, January 2010, pp. 61–87, DOI [10.3233/JCS-2010-0373](https://doi.org/10.3233/JCS-2010-0373)
- [56] Q. Ruihong and M. Ying, “Research Progress Of Quantum Repeaters”, *Journal of Physics: Conference Series*, vol. 1237, June 2019, pp. 1–7, DOI [10.1088/1742-6596/1237/5/052032](https://doi.org/10.1088/1742-6596/1237/5/052032)
- [57] J. Jun, “Squeezing Effect on Entanglement Swapping: A Quantum Trajectory Approach”, *Journal of the Korean Physical Society*, vol. 73, October 2018, pp. 1025–1028, DOI

- 10.3938/jkps.73.1025
- [58] F. Xu, X. Ma, Q. Zhang, H.-K. Lo, and J.-W. Pan, “Secure quantum key distribution with realistic devices”, *Review of Modern Physics*, vol. 92, May 2020, pp. 1–68, DOI [10.1103/RevModPhys.92.025002](https://doi.org/10.1103/RevModPhys.92.025002)
- [59] B. Qi, C.-H. F. Fung, H.-K. Lo, and X. Ma, “Time-Shift Attack in Practical Quantum Cryptosystems”, *Quantum Information and Computation*, vol. 7, January 2007, pp. 73–82. <https://arxiv.org/abs/quant-ph/0512080>
- [60] C.-H. F. Fung, K. Tamaki, B. Qi, H.-K. Lo, and X. Ma, “Security proof of quantum key distribution with detection efficiency mismatch”, *Quantum Information and Computation*, vol. 9, March 2008, pp. 131–165. <https://arxiv.org/abs/0802.3788>
- [61] P. C. Consul and G. C. Jain, “A Generalization of the Poisson Distribution”, *Technometrics*, vol. 15, November 1973, pp. 791–799, DOI [10.1080/00401706.1973.10489112](https://doi.org/10.1080/00401706.1973.10489112)
- [62] C. Branciard, N. Gisin, B. Kraus, and V. Scarani, “Security of two quantum cryptography protocols using the same four qubit states”, *Physical Review A*, vol. 72, September 2005, pp. 1–18, DOI [10.1103/physreva.72.032301](https://doi.org/10.1103/physreva.72.032301)
- [63] Y. Fei, X. Meng, M. Gao, H. Wang, and Z. Ma, “Quantum man-in-the-middle attack on the calibration process of quantum key distribution”, *Scientific Reports*, vol. 8, March 2018, pp. 1–10, DOI [10.1038/s41598-018-22700-3](https://doi.org/10.1038/s41598-018-22700-3)
- [64] H.-K. Lo, M. Curty, and B. Qi, “Measurement-Device-Independent Quantum Key Distribution”, *Physical Review Letters*, vol. 108, March 2012, pp. 1–5, DOI [10.1103/physrevlett.108.130503](https://doi.org/10.1103/physrevlett.108.130503)
- [65] N. Gisin, S. Fasel, B. Kraus, H. Zbinden, and G. Ribordy, “Trojan-horse attacks on quantum-key-distribution systems”, *Physical Review A*, vol. 73, February 2006, pp. 1–6, DOI [10.1103/PhysRevA.73.022320](https://doi.org/10.1103/PhysRevA.73.022320)
- [66] P. V. P. Pinheiro, P. Chaiwongkhot, S. Sajeed, R. T. Horn, J.-P. Bourgoin, T. Jennewein, N. Lütkenhaus, and V. Makarov, “Eavesdropping and countermeasures for backflash side channel in quantum cryptography”, *Optics Express*, vol. 26, August 2018, pp. 21020–21032, DOI [10.1364/OE.26.021020](https://doi.org/10.1364/OE.26.021020)
- [67] A. Huang, R. Li, V. Egorov, S. Tchouragoulov, K. Kumar, and V. Makarov, “Laser-Damage Attack Against Optical Attenuators in Quantum Key Distribution”, *Physical Review Applied*, vol. 13, March 2020, pp. 1–14, DOI [10.1103/PhysRevApplied.13.034017](https://doi.org/10.1103/PhysRevApplied.13.034017)
- [68] C. Elliott, D. Pearson, and G. Troxel, “Quantum cryptography in practice”, *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Karlsruhe (Germany), August 2003, pp. 227–238, DOI [10.1145/863955.863982](https://doi.org/10.1145/863955.863982)
- [69] N. Lutkenhaus, “Estimates for practical quantum cryptography”, *Physical Review A*, vol. 59, May 1999, pp. 3301–3319, DOI [10.1103/PhysRevA.59.3301](https://doi.org/10.1103/PhysRevA.59.3301)
- [70] A. N. Bugge, S. Sauge, A. M. M. Ghazali, J. Skaar, L. Lydersen, and V. Makarov, “Laser damage helps the eavesdropper in quantum cryptography”, *Physical review letters*, vol. 112, February 2014, pp. 1–5, DOI [10.1103/PhysRevLett.112.070503](https://doi.org/10.1103/PhysRevLett.112.070503)
- [71] H.-K. Lo, M. Curty, and B. Qi, “Measurement-device-independent quantum key distribution”, *Physical review letters*, vol. 108, March 2012, pp. 1–5, DOI [10.1103/PhysRevLett.108.130503](https://doi.org/10.1103/PhysRevLett.108.130503)
- [72] Qiskit, <https://qiskit.org/>
- [73] J. Dehaene and B. De Moor, “Clifford group, stabilizer states, and linear and quadratic operations over GF(2)”, *Physical Review A*, vol. 68, October 2003, pp. 1–10, DOI [10.1103/PhysRevA.68.042318](https://doi.org/10.1103/PhysRevA.68.042318)
- [74] D. Jaschke, M. L. Wall, and L. D. Carr, “Open source Matrix Product States: Opening ways to simulate entangled many-body quantum systems in one dimension”, *Computer physics communications*, vol. 225, April 2018, pp. 59–91, DOI [10.1016/j.cpc.2017.12.015](https://doi.org/10.1016/j.cpc.2017.12.015)
- [75] V. Rajaraman, “Cloud computing”, *Resonance*, vol. 19, April 2014, pp. 242–258, DOI [10.1007/s12045-014-0030-1](https://doi.org/10.1007/s12045-014-0030-1)
- [76] K. Kirkpatrick, “Software-defined networking”, *Communications of the ACM*, vol. 56, September 2013, pp. 16–19
- [77] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, “Network function virtualization: state-of-the-art and research challenges”, *IEEE Communicatino Surveys and Tutorials*, vol. 18, September 2015, pp. 236–262, DOI

- [10.1109/COMST.2015.2477041](https://doi.org/10.1109/COMST.2015.2477041)
- [78] S. Nauerth, F. Moll, M. Rau, J. Horwath, S. Frick, C. Fuchs, and H. Weinfurter, “Air to ground quantum key distribution”, *Quantum Communications and Quantum Imaging X*, San Diego (CA, USA), October 25, 2012, pp. 71–76, DOI [10.1117/12.929790](https://doi.org/10.1117/12.929790)
- [79] H. Wu, F. Bao, D. Ye, and R. Deng, “Cryptanalysis of Polynomial Authentication and Signature Scheme”, *ACISP: Australasian Conference on Information Security and Privacy*, Brisbane (Australia), July 10-12, 2000, pp. 278–288, DOI [10.1007/10718964\\_23](https://doi.org/10.1007/10718964_23)

# Appendix A

## User Manual

This chapter explains the steps needed to configure and run the proposed work, analysing the main implementation choices. This work is basically composed of two main implementations: the QKD simulator and the QKD Key server. These implementations consist of different docker containers that are orchestrated through a docker compose file.

### A.1 QKD Simulator

QKD simulator is composed of 7 different containers. Each container represents an instance or a service needed in the implementation. All containers can be run simultaneously by executing a docker compose file with the following commands:

```
docker-compose build
docker-compose up
```

Docker compose used is version 2.1. The configuration file sets up the following containers:

- A mysql container, named *DBSim* with address 172.15.0.2
- A vault container, named *vaultSim* with address 172.15.0.3
- A container running a simulator instance, named *AliceSim* with address 172.15.0.4
- A container running another simulator instance, named *BobSim* with address 172.15.0.5
- A container running both classical and quantum channel instances, named *channelSim* with address 172.15.0.6
- A container running singlet source instance, named *singletSource* with address 172.15.0.7
- A container running simulator manager instance, named *simulatorManager* with address 172.15.0.8

The setup of each single container will here be outlined.

#### mysql

mysql container is the default mysql image in the docker hub. An instance of this server can be run by issuing the following command:

```
docker run -e MYSQL_ROOT_PASSWORD=xxxxx
```

More information regarding configuration of this docker can be found in the official documentation <sup>1</sup>. If this docker has been set up through the provided docker compose file, this command can be issued in order to start it:

```
docker start DBSim
```

A similar command can be issued if this container needs to be stopped:

```
docker stop DBSim
```

Current configuration sets the password to `dummyspassword` and select the database called `simulatorDB`. Configuration for this database is loaded from a sql file that creates the following tables:

```
CREATE TABLE alice_sim_DB (  
  requestIP varchar(255) NOT NULL,  
  complete tinyint(1) DEFAULT NULL,  
  exchangedKey text,  
  verified tinyint(1) DEFAULT NULL,  
  PRIMARY KEY (requestIP)  
)
```

```
CREATE TABLE bob_sim_DB (  
  requestIP varchar(255) NOT NULL,  
  complete tinyint(1) DEFAULT NULL,  
  exchangedKey text,  
  verified tinyint(1) DEFAULT NULL,  
  PRIMARY KEY (requestIP)  
)
```

```
CREATE TABLE aliceSim_pre_keys (  
  keyID varchar(255) NOT NULL,  
  TTL int NOT NULL,  
  PRIMARY KEY (keyID)  
)
```

```
CREATE TABLE bobSim_pre_keys (  
  keyID varchar(255) NOT NULL,  
  TTL int NOT NULL,  
  PRIMARY KEY (keyID)  
)
```

This configuration is loaded from the docker compose to ensure that `simulatorDB` database and above tables are already available when simulator starts its execution.

## Vault

Vault server as well has been developed starting from the official image in docker hub. The image can be run by issuing the following command:

```
docker run --cap-add=IPC_LOCK -e 'VAULT_LOCAL_CONFIG={"backend": {"file":  
  {"path": "/vault/file"}}, "default_lease_ttl": "168h", "max_lease_ttl":  
  "720h"}' vault server
```

More information about docker configuration are available in the official docker hub documentation <sup>2</sup>. If docker compose has been used to setup this docker it is possible to start the container with the following command:

---

<sup>1</sup>[https://hub.docker.com/\\_/mysql](https://hub.docker.com/_/mysql)

<sup>2</sup>[https://hub.docker.com/\\_/vault](https://hub.docker.com/_/vault)

```
docker start vaultSim
```

Vault container can be stopped with this command:

```
docker stop vaultSim
```

After vault server is run, it must be initialised. Initialisation is done by connecting vault server through the vault client CLI. The first time vault is started it should be initialised with the command:

```
vault operator init
```

This command returns a set of unseal keys together with the root token. Unsealing keys should be used every time vault server is restarted in order to allow data modification. Root token will be used inside the code to authenticate request made to vault.

Once vault is initialised, generated keys and secure data are saved in folders volumes/file and volumes/config of the hosting system. Hence, when vault container is started again, the status is maintained and initialisation is no longer required. With the current configuration, in order to unseal vault when it is started again the following commands should be issued:

```
vault operator unseal
62cbf8b7c181822c774d95d6ae5bd29d8f97f642f30cdcb6cc511869447131dcf8
vault operator unseal
d818f6a40a94882d78feb4fbf84f89e2b29a5ef6e45d3b3beae4dfc08a06351d63
```

For further information about vault configuration, refer to vault official documentation <sup>3</sup>.

## Simulator instances

Two simulator instances needs to be set up in order to exchange keys between two parties. The two instances share the same source code, however some settings need to be different for the two instances. Specific settings for each instance are located in a yaml configuration file. Inside docker compose the two configuration files are mapped in the same folder of the container, so that no modification to the source code need to be done.

Since simulator source code is pure python, docker containers for simulator are two python container running python version 3.7. Python containers are launched through a Dockerfile that install module requirements and run the simulator source code. Even python docker image has been downloaded from the official docker hub image. More information about this image are available in docker hub documentation documentation <sup>4</sup>.

By using provided docker compose file, it is possible to start the container instances running simulator implementation with the following commands:

```
docker start AliceSim
docker start BobSim
```

The same instances can instead be stopped in this way:

```
docker stop AliceSim
docker stop BobSim
```

Interfaces provided by simulator can be seen in table [A.1](#).

Web interface is accessible through port 4000. Methods grouped in BB84 interface and E91 interface are the ones used by the two simulator instances to communicate each other. They are

---

<sup>3</sup><https://www.vaultproject.io/docs>

<sup>4</sup>[https://hub.docker.com/\\_/python](https://hub.docker.com/_/python)

<i>Web interface</i>		
Method Name	URL	Access Method
<i>BB84 interface</i>		
getQuantumKey	http://{simulator_IP}/sendRegister	POST
compareBasis	http://{simulator_IP}/compareBasis	POST
verifyKey	http://{simulator_IP}/verifyKey	POST
<i>E91 interface</i>		
getBasis	http://{simulator_IP}/getBasis	GET
setKey	http://{simulator_IP}/setKey	POST
<i>General interface</i>		
startKeyExchange	http://{simulator_IP}/startKeyExchange	POST
settings	http://{simulator_IP}/settings	POST
<i>Module interface</i>		
Method Name	Parameters	
begin()	serverPort	
exchangeKey()	key_length, destination, protocol, timeout	
end()		

Table A.1.

used to realise the related QKD protocol and their usage has already been described in chapter 8.

Methods grouped under general interface and module interface can be used by high level applications, as well as simulator manager, to send commands to a simulator instance. The method `exchangeKey` from module interface can be mapped to the method `startKeyExchange` from the web interface. They both provide an access to the core functionality of the simulator, by returning a quantum exchanged key. If in `exchangeKey` method parameters are specified in the function prototype, `startKeyExchange` method allows to select specific configuration in the body of the POST request. For example a key exchange from web interface can be required from python with the following instruction:

```
requests.post('http://172.15.0.4:4000/startKeyExchange', data =
    repr({'destination' : 'http://172.15.0.5:4000', 'length' : 128,
        'protocol' : 'BB84'}))
```

Web interface also defines `settings` method to change authentication method on the classical channel and statevector length. This method belongs to web interface only and there is no way to change these settings from module interface. `begin` and `end` functions instead only belong to module interface. In this implementation they are used to initialise web interface, but they can be used to perform attach and detach operation for a real device.

## Channel

Quantum and classical channel are designed with a docker container that forward all requests from source to destination. Channel is just another python container that execute a python program with a Flask web service. Channel container is set up through a dedicated Dockerfile that is located in a different folder from the Dockerfile used for the simulator instances. In this way it is possible to manage it form a single docker compose file.

As for simulator, Dockerfile for channel simply installs module requirements and execute the channel source code.

Commands to start and stop this container after the initial docker compose set up are the following:

```
docker start channelSim
docker stop channelSim
```

Channel is accessible only through a web interface available on port 4000. This interface is composed of the methods described by table A.2. The first three methods of this interface are

Method Name	URL	Access Method
forwardQubits	http://{channel_IP}/sendRegister	POST
compareBasis	http://{channel_IP}/compareBasis	POST
verifyKey	http://{channel_IP}/verifyKey	POST
toggleAttack	http://{channel_IP}/attacks	POST
startE91exchange	http://{channel_IP}/startE91exchange	POST

Table A.2.

used by BB84 protocol. In particular each instance of simulator uses this interface to send data to the other instance. Channel can simply forwards incoming data or manipulating them before forwarding depending on the global settings. Destination to which forward the data, has to be defined as query parameter in each of these methods and has to be specified every time one of these methods is called.

`toggleAttack` method is used to enable or disable attacks on channel. Attacks should be specified in query parameters and the currently available parameters are `interceptAndResend` and `manInTheMiddle`. An example call that enable intercept and resend attack is the following:

```
requests.post('http://172.15.0.6:4000/attacks?interceptAndResend=1')
```

`startE91exchange` is the method that simulator instances call to start a key exchange with E91 protocol. Once this method is called, it performs the steps already described in chapter 8 which can be seen in figure 8.9.

### Singlet source

Singlet source is run from a dedicated docker container. Its source code is located in a dedicated folder in order to be managed by a different Dockerfile. As for the previous case this Dockerfile install needed requirements and execute the source code in a python 3.7 container.

When docker compose is used, the following commands can be used to start and stop the container:

```
docker start singletSource
docker stop singletSource
```

Even singlet source web interface is accessible from a web interface only, available on port 4000. This interface is composed of just one method that allows to require a desired number of singlets. This interface is used only by channel during E91 protocol. An example call to request 1024 singlets is the following:

```
requests.get('http://172.15.0.7:4000/getQbits?number=1024')
```

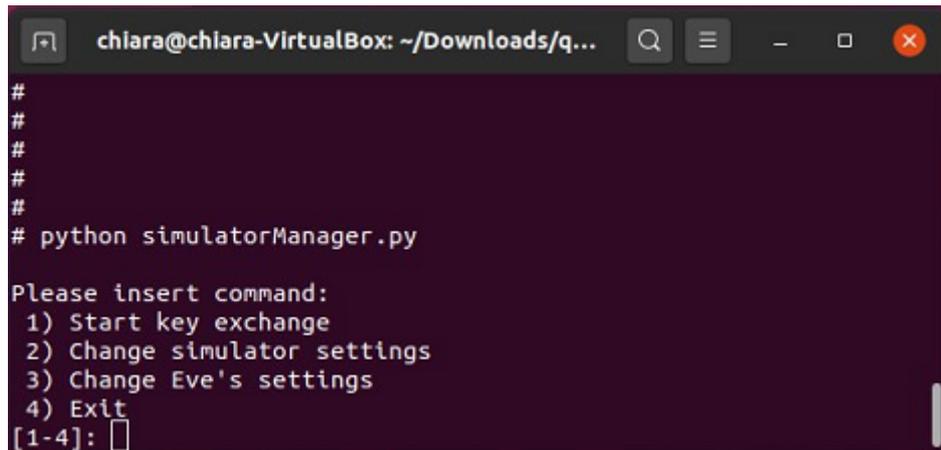
### Simulator manager

Simulator manager is another instance of python 3.7 container. Its source code is located in a dedicated folder. Manager can be interfaced by accessing its console or by reaching its web interface. In order to access manager console the following command can be issued:

```
docker-compose exec manager sh
```

This command will give access to the shell interface of the docker container. From this interface it is possible to launch the manager command line interface with the following command:

```
python simulatorManager.py
```

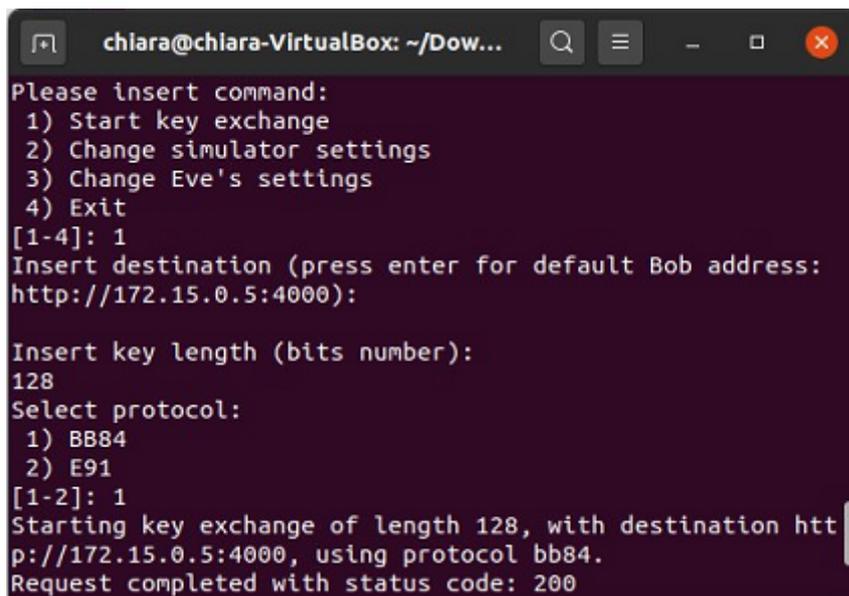


```
chiara@chiara-VirtualBox: ~/Downloads/q...
#
#
#
#
# python simulatorManager.py
Please insert command:
 1) Start key exchange
 2) Change simulator settings
 3) Change Eve's settings
 4) Exit
[1-4]:
```

Figure A.1. Simulator manager command line interface.

Command line interface allows to send commands to the whole system. As shown by figure A.1 the manager interface allows to select among 4 different operations.

1. By selecting option 1, it is possible to start a new key exchange. The manager will automatically reach AliceSim web interface to start the key exchange. However, after selecting the first option, a set of configuration can still be selected. In particular it is possible to select the destination of the key exchange, however the default destination is BobSim web interface and its address is already hardcoded inside the manager. Hence if the docker compose file is used without modifications, it is enough to skip destination selection by pressing enter button without specifying any destination. After destination has been chosen, manager will require the desired key length and the protocol that will be used during the exchange. As said in previous chapters, the current available protocols are BB84 and E91. A detail of the command line interface during this operation can be seen in figure A.2. Note that man-



```
chiara@chiara-VirtualBox: ~/Dow...
Please insert command:
 1) Start key exchange
 2) Change simulator settings
 3) Change Eve's settings
 4) Exit
[1-4]: 1
Insert destination (press enter for default Bob address:
http://172.15.0.5:4000):
http://172.15.0.5:4000):

Insert key length (bits number):
128
Select protocol:
 1) BB84
 2) E91
[1-2]: 1
Starting key exchange of length 128, with destination htt
p://172.15.0.5:4000, using protocol bb84.
Request completed with status code: 200
```

Figure A.2. Simulator manager CLI for key exchange.

ager behaves as high level application for both the simulator instances hence, after having required the key to Alice interface, it requires the same key to Bob web interface in order to simulate a normal flow between two applications that need to access the same key.

2. By selection option 2 it is possible to change settings inside simulator. These settings are the statevector length (as already described in chapter 11), and authentication method (selecting between AES-GCM and *SPHINCS<sup>+</sup>*). Even in this case, manager takes care of changing the desired setting in both of the two instances of the simulator.
3. Option 3 allows to change channel settings. In particular it allows to select the behaviour the eavesdropper will use during the subsequent exchanges. After selecting this option, the manager will ask if currently available attacks need to be enabled or not and will send a request to the channel web interface in order to fulfil the request. The currently available attacks are *Intercept and Resend* that consists of measuring quantum bits before forwarding them to destination and *Man In The Middle* that refers to attacks on classical channel but is currently not implemented.

Web interface provides another access to the same operations. In order to access it, it is enough to reach its address from a web browser. In this case the following address needs to be specified:

```
http://172.15.0.8:4000
```

Web interface is a python Flask server running a Flask Black Dashboard. This interface has already been shown in figure 8.6. In order to design the desired interface the standard Flask Black Dashboard example has been modified to meet simulator requirements. In particular graphical interface has been designed by modifying the index.html file under manager/app/home/templates folder. The javascript source code associated to the html file is located under manager/app/base/static/assets/js/ui.js file.

Simulator manager container has also another important role. Has soon as it is started, it waits for vault and mysql servers to be up and then automatically unseals vault storage and adds preshared keys and their reference to mysql database in order to use them for AES-GCM authentication. If manager container is not started these operations need to be done manually.

## A.2 QKD Key Server and Module

This implementation regards the integration of a QKD system in a softwarised network. A set of different docker containers has been used in order to simulate a distributed environment. As for QKD Simulator, these containers too were orchestrated through a docker compose file that allows to run all containers simultaneously. In order to start configuration included in docker container file the following commands must be issued:

```
docker-compose build
docker-compose up
```

The configuration file sets up the following containers:

- A mysql container, named *DBStorage* with address 172.17.0.2
- A keycloak container, named *Authentication* with address 172.17.0.3
- A vault container, named *vaultAlice* with address 172.17.0.4
- A vault container, named *vaultBob* with address 172.17.0.5
- A container running the whole architecture with QKD Key Server, QKDM and simulator, named *AliceServer* with address 172.17.0.6
- A container running the whole architecture with QKD Key Server, QKDM and simulator, named *BobServer* with address 172.17.0.7

All involved containers for this implementation will be here discussed.

**mysql**

mysql container can be initialised as already discussed in previous section. The docker is here initialised with a different sql file that creates two different database, one for each instance, containing the following tables:

```
CREATE TABLE KmeExchangerData (  
  KME_ID varchar(255) NOT NULL,  
  key_handle varchar(255) DEFAULT NULL,  
  key_ID varchar(255) NOT NULL,  
  open tinyint(1) NOT NULL,  
  module_ID varchar(255) DEFAULT NULL,  
  module_address varchar(255) DEFAULT NULL  
)
```

```
CREATE TABLE bb (  
  requestIP varchar(255) NOT NULL,  
  complete tinyint(1) DEFAULT NULL,  
  exchangedKey text,  
  verified tinyint(1) DEFAULT NULL,  
  PRIMARY KEY (requestIP)  
)
```

```
CREATE TABLE connectedSAE (  
  SAE_ID varchar(255) NOT NULL,  
  SAE_IP varchar(255) NOT NULL,  
  PRIMARY KEY (SAE_ID)  
)
```

```
CREATE TABLE currentExchange (  
  destination varchar(255) NOT NULL,  
  handle varchar(255) NOT NULL,  
  PRIMARY KEY (destination)  
)
```

```
CREATE TABLE destinations (  
  SAE_ID varchar(255) NOT NULL,  
  KME_ID varchar(255) NOT NULL,  
  PRIMARY KEY (SAE_ID)  
)
```

```
CREATE TABLE exchangedKeys (  
  KME_ID varchar(255) NOT NULL,  
  KME_IP varchar(255) NOT NULL,  
  KME_PORT int NOT NULL,  
  AUTH_KEY_ID varchar(255) NOT NULL,  
  KEY_IDs text,  
  KEY_COUNT int DEFAULT '0',  
  DEF_KEY_SIZE int DEFAULT '128',  
  MAX_KEY_COUNT int DEFAULT '500',  
  MAX_KEY_PER_REQUEST int DEFAULT '1',  
  MAX_KEY_SIZE int DEFAULT '1024',  
  MIN_KEY_SIZE int DEFAULT '8',  
  MAX_SAE_ID_COUNT int DEFAULT '0',  
  PRIMARY KEY (KME_ID)  
)
```

```
CREATE TABLE handles (  

```

```
    handle varchar(255) NOT NULL,  
    destination varchar(255) NOT NULL,  
    timeout int DEFAULT NULL,  
    length int DEFAULT NULL,  
    synchronized tinyint(1) NOT NULL,  
    PRIMARY KEY (handle)  
)  
  
CREATE TABLE log (  
    timestamp timestamp NULL DEFAULT CURRENT_TIMESTAMP,  
    level int NOT NULL,  
    message text NOT NULL  
)  
  
CREATE TABLE qkdmodules (  
    moduleID varchar(255) NOT NULL,  
    module varchar(255) NOT NULL,  
    protocol varchar(255) DEFAULT NULL,  
    moduleIP varchar(255) NOT NULL,  
    PRIMARY KEY (moduleID)  
)  
  
CREATE TABLE reservedKeys (  
    KME_ID varchar(255) NOT NULL,  
    SAEKeys text,  
    PRIMARY KEY (KME_ID)  
)
```

## Vault

Vault is configured as already described in the previous version, with a volume folder representing storage and configuration the server provides. The difference with the previous case is that this time two different vault containers are used, rather than having the two instances writing on different paths of the same vault container. This was done to properly simulate a distributed environment in which each instance has its own services which differ from the ones of the others. From the implementation point of view, having two different vault servers or using the same container does not involve any changes. Inside the source code, the pointer to vault server is realised through an entry on the yaml configuration file. Hence, it is enough to change the address of vault server inside the configuration file to let each instance point to the desired server. Thus, the choice of having different vault servers or just one is transparent for the implementation.

If provided volumes folder are used, the two vault server are already initialised. In such a case one instance of vault server can be unsealed with the following commands:

```
vault operator unseal  
    62cbf8b7c181822c774d95d6ae5bd29d8f97f642f30cdcb6cc511869447131dcf8  
vault operator unseal  
    d818f6a40a94882d78feb4fbf84f89e2b29a5ef6e45d3b3beae4dfc08a06351d63
```

While the other instance can be unsealed by typing these commands from a command line interface:

```
vault operator unseal  
    0aa3e6372ca61d6a232f30aadf70e489bbc45787761f89ca8564b2816c98ae9b8d  
vault operator unseal  
    f67b2d8eee2f193b046081cb94e45b9b2bbbed6996155102dc1ec6676459fa9ffec
```

## Keycloak

Keycloak is the server used to authenticate requests coming from high level security application in QKD Key Server. Keycloak container can be launched by issuing the following command:

```
docker run -p 8080:8080 -e KEYCLOAK_USER=admin -e KEYCLOAK_PASSWORD=admin
quay.io/keycloak/keycloak:11.0.3
```

Further information about the configuration of this container can be found in the official docker hub documentation <sup>5</sup>.

After Keycloak is running it is possible to connect to its web interface in order to create an *authentication realm* and register clients. Keycloak web interface is available through a web browser at address:

```
http://172.17.0.3:8080
```

After accessing the interface it is possible to add a new realm as showed by figure A.3. In this implementation the realm `quantum_auth` has been defined. From the same interface it is

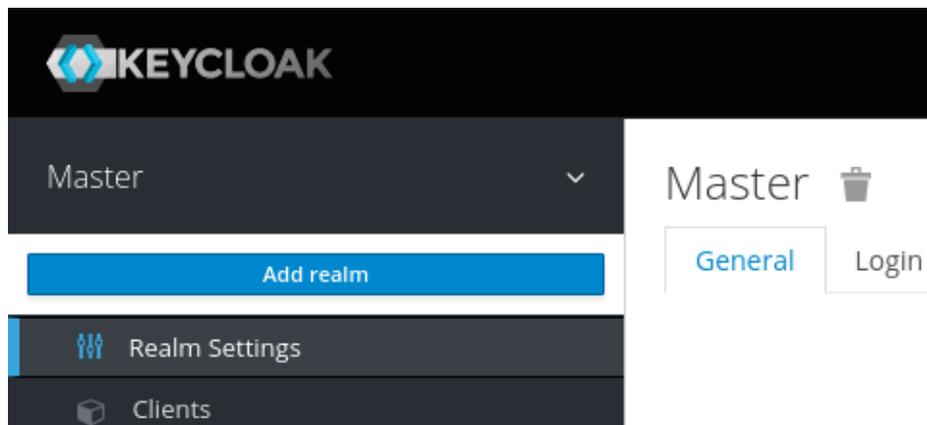


Figure A.3. Keycloak web interface.

possible to register new clients which represent the high level security applications that needs to be authenticated inside QKD Key server. Clients must be registered with *openid-connect* protocol in order to use an authentication token when accessing key server. Client configuration can be seen from figure A.4

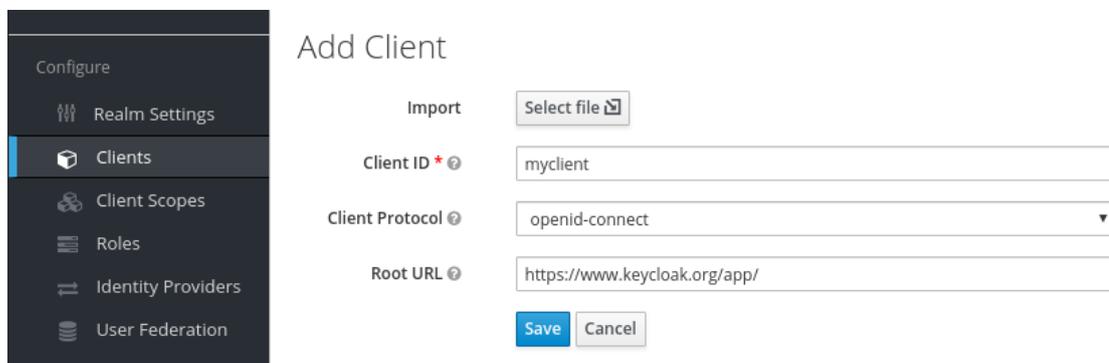


Figure A.4. Keycloak client settings.

<sup>5</sup><https://hub.docker.com/r/jboss/keycloak>

Key server instances should be registered as client as well in order to get access to keycloak server and validate received token. High security applications which want to access key server API, should obtain an authentication token from keycloak with a POST request to the following endpoint:

`http://172.17.0.3:8080/auth/realms/quantum_auth/protocol/openid-connect/token`

The request should contain client ID and a client secret in its body in order to be successful. With the retrieved token, high level security applications can access key server by specifying the token with the following header inside the request:

**Authorization:** Bearer <token>

Key server will check the token before parsing each request. It will forward the token to keycloak server to make sure the token belong to a valid application that is now being authenticated. If keycloak does not validate the token, it will return an error code that will cause the key server to decline the request. All this mechanism is managed automatically by using `OpenIDConnect` class from `flask_oidc` module in the key server source code. With this class it is possible to add a decorator before each request which requires authentication, to manage all these operations in background, before the code to manage the request is called. The needed decorator is the following:

`@oidc.accept_token(True)`

Further information about keycloak configuration can be found in the official documentation <sup>6</sup>

### Key server and QKD Module

Two instances of this container are used to represent two parties involved in a communication. Even more than two instances can be set up in order to represent a more complex network. For each instance it is enough to change the yaml configuration file in order to add the specific settings. The docker container in a python3 docker as it is for QKD Simulator. In this case the container executes the code for the Key server and all selected instances of QKD module (which can be more than one for each server).

<i>Web interface</i>		
Method Name	URL	Access Method
open	<code>http://{QKDM_IP}/open</code>	POST
start	<code>http://{QKDM_IP}/start</code>	POST
<i>Module interface</i>		
Method Name	Parameters	
OPEN_CONNECT()	source, destination, QOS, Key_stream_ID, status	
GET_KEY()	Key_stream_ID, index, Key_buffer, Metadata, status	
CLOSE()	Key_stream_ID, status	

Table A.3.

Even in this case a Dockerfile is used to correctly initialise the system. The Dockerfile take cares of installing required python modules and prepare the environment before Key Server is launched. After that, it starts Key Server and required QKD Module instances with the following commands:

<sup>6</sup><https://www.keycloak.org/documentation>

```
python KeyServer.py <serverPort>
python QKModule.py <serverPort>
```

QKD Module provides two interfaces. The first one is a web interface accessible at port 4000 and is used by the QKDM to communicate with its peer. The second one is a module interface that a high level module (key server in this case) can use to interact with it. These interfaces provide the methods depicted by table A.3.

QKD Key server is composed by multiple interfaces. Most of the methods are defined in the web interface that is accessible through port 5000. A simple module interface defines a single method that allows to attach QKDMs to the system. All key server methods are described by table A.4.

<i>Web interface</i>		
Method Name	URL	Access Method
<i>Northbound Interface</i>		
getStatus	https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/status	GET
getKey	https://{KME_hostname}/api/v1/keys/{slave_SAE_ID}/enc_keys	POST
getKeyWithKeyIDs	https://{KME_hostname}/api/v1/keys/{master_SAE_ID}/dec_keys	POST
getPreferences	https://{KME_hostname}/api/v1/preferences	GET
setPreference	https://{KME_hostname}/api/v1/preferences/{preference}	POST
getInfo	https://{KME_hostname}/api/v1/information/{info}	GET
<i>Internal web Interface</i>		
challenge	http://{KME_hostname}/api/v1/challenges/kme	GET
authenticate	http://{KME_hostname}/api/v1/authenticate/kme	POST
knownSAE	https://{KME_hostname}/api/v1/saes/{slave_SAE_ID}	GET
reserveKeys	https://{KME_hostname}/api/v1/kids/{master_SAE_ID}	POST
openRequest	https://{KME_hostname}/api/v1/keys/	POST
syncRequest	https://{KME_hostname}/api/v1/keys/key	POST
<i>Module interface</i>		
Method Name	Parameters	
<i>Southbound Interface</i>		
QKD_REGISTER_MODULE()	qkdModule, protocol, address	

Table A.4.

# Appendix B

## Developer's Manual

In this chapter the main implementation choices will be discussed, outlining the main libraries that led to the proposed implementation.

### B.1 QKD Simulator

As said the whole implementation uses python 3.7. The whole code of QKD simulator can be found in *qkd-simulator* folder. This folder contains the simulator source code (in *Simulator.py* file), the true random number implementation (in *trng.py* file) and all needed files to configure the system. The folder also contains other folders that represents the other python containers (channel, manager and singlet source), plus a *volumes* folder that contains the storage drives some containers will have attached during execution.

All these containers are orchestrated through a docker compose file (*docker-compose.yml*) that sets up the configuration already discussed in previous chapters. This file will generate a dedicated subnetwork in which all dockers will operate. The subnetwork is defined as follows:

```
networks:
  frontend:
    ipam:
      config:
        - subnet: 172.15.0.0/24
```

Inside docker compose file, storage drives are managed through *volumes* property. Here each entry represents the source of the file or folder to be attached and, split by ":" the target destination where the source will reside inside the container. For vault container, volumes property is defined as follows:

```
volumes:
  - ./volumes/logs:/vault/logs
  - ./volumes/file:/vault/file
  - ./volumes/config:/vault/config
```

Where *./volumes* is the local folder contained in the root folder of the source code, while */vault/* is the destination folder inside the container in which selected folder will be mounted.

Mysql container uses volumes property to copy the sql configuration file, that will be used to initialise the database. The setting is the following:

```
volumes:
  - ./db_init.sql:/data/application/db_init.sql
```

Simulator instances use volumes property to obtain a different configuration file (*configS.yaml*). In docker compose file this is represented by the following settings:

```
alice:
  build: .
  volumes:
    - ./volumes/alice:/usr/src/app/config

bob:
  build: .
  volumes:
    - ./volumes/bob:/usr/src/app/config
```

As can be seen from this snippet, both Alice and Bob containers run the code located in the root directory. `build: .` indicates docker compose to build container starting from the Dockerfile located in the same folder as the docker-compose.yml. The only difference between the two container is the configuration file, that is stored in the same destination directory (`/usr/src/app/config`), but with a different source file (`./volumes/alice` for Alice container and `./volumes/bob` for Bob container).

Channel, singlet source and simulator manager container do not lean on volumes property.

All containers use the same property to select the restart option of each container. This property is defined as follows:

```
restart: "on-failure"
```

Indicating that if a container stops its execution for a runtime error, the container is automatically restarted.

The last settings that is worth to mention regards simulator manager container. This container is in charge of unsealing vault storage and load preshared keys for AES-GCM authentication method. These operations are performed as soon as container starts. Hence, it is important to make sure that vault and mysql containers are already running when manager tries to access them. This is done with the following configuration that ensure both of the containers are started before manager starts its execution:

```
depends_on:
  - vault
  - mysql
```

## Simulator instances

As said simulator source code is located in Simulator.py file. The main library used here is qiskit that allows to build and run quantum circuit in a simulated environment. A basic explanation on how to simulate quantum circuit with python qiskit module has already been provided in chapter 6. Here a more detailed discussion about how qiskit has been used will be provided.

Qubits in python environment are manipulated through Statevector. This object is capable of mathematically defines the state of a qubit (or a set of qubits) and evolve this state depending by the gates it passes through. Qubits can reside in QuantumRegister too. However a QuantumRegister object does not maintain its state among measurements performed with different circuits. Hence Statevector was the choice for modelling quantum bits in this implementation. The object is also easily serialisable in order to be sent over a network, without losing its properties.

The drawback of Statevector object is that it is capable of managing only 14 qubits at time. A Statevector with more than 14 qubits will cause memory error. Hence, when simulator needs to send a given number of qubits to exchange a key with a given length, it splits the required length in chunks of a length comprised by 1 and 14 and sends an array of Statevector to the other instance. Statevector length is adjustable from simulator settings as has been shown in previous chapter. However, due to the very low number of qubits a Statevector can manage, data over quantum channel will always be an array of objects.

Another thing to notice about Statevector regards the way the measurements result should be read. A Statevector can be created and measured with the following instructions:

```
statevector = Statevector.from_label('00000000')
resulting_statevector = statevector.evolve(quantum_circuit)
measurement_results = resulting_statevector.measure()[0]
```

`evolve` function let the state of qubits pass through a quantum circuit, by applying the gates belonging to it. If the quantum circuit has a given gate applied on the first qubit, when using `evolve` function, this gate will be applied to the last one. Basically measurement result are flipped with respect to a normal measurement. Hence, inside the source code, flip operations are performed in order to make sure that each bit corresponds to the one passed to the gate selected with the basis table.

Authentication over classical channel can be done exploiting *SPHINCS*<sup>+</sup> or AES-GCM. For both of them a different python library has been used. *SPHINCS*<sup>+</sup> is realised through `pyspx` module, while AES-GCM exploits `cryptography` module. As said, AES-GCM uses some preshared keys that are loaded beforehand by simulator manager container. Right now, there is no way to reload preshared key from the simulator itself. If more keys are needed, these should be loaded manually as done by manager. For *SPHINCS*<sup>+</sup> instead a public and private key pair has been used. This key is written in the `yaml` configuration file of the simulator. However, this key never changes (unless a modification to `yaml` file is done). This can represent a security concern in a real environment since *SPHINCS*<sup>+</sup> protocol grants security only if the same key is used for a limited number of times, since every signature leaks some information related to the key. This problem has not been currently managed and the only way to use different authentication keys is to manually change them in `yaml` configuration file.

Multiple signature process within the same key exchange can affect the time required by a key exchange. In order to make sure signature time will give the minimum contribution possible to the overall time, BB84 protocol has been designed to send data over classical channel just one time per exchange, allowing to perform just one signature for each key exchange. The idea is to try to calculate the required number of qubits before the key is actually exchanged. After that, qubits are sent to the simulator peer through the quantum channel. At this point key sifting needs to take place. In order to complete this step simulator instances need to exchange their basis table over classical channel. Since all qubits were sent beforehand, the exchange here regards the basis table related to all those qubits, hence a single signature is needed. If, at the end of key sifting procedure, the number of bits composing the key is not enough, the whole key is discarded and the exchange needs to be performed again. This allows to significantly reduce time associated to signature since just one signature takes place. However, it becomes crucial to correctly estimate the number of qubits that will be needed. Since BB84 uses just two basis, there is the 50% of chance that the two simulator instances measure a qubit in the same basis. This translates into the need of exchanging the double length of the key length since half of them will be discarded due to measure in different basis. Hence statistically, for a 128 bits key, 256 qubits need to be exchanged. However, this is just a statistical calculation, in a real exchange a slight variation from the 50%. In order to make sure the correct number of qubits is always exchanged, simulator calculates qubits length as the double length of the key, plus a 15% of this length to secure from probability variations.

Configuration file for this instance allows to change the settings here described:

```
auth_key:
  peerPublicKey:
    b'64ewf98wqrsdfft1^\xbf\x9a\x1e\xdc\xac+\x94\x06E\x12\xfa?\xa2\xddf'
  privateKey: b'124986546848776451342111546854654643545484981234>\xff
    \xb3\xa9\x96F\x94\xbc\xadHz\xca\xcd\xc7+.'
```

```
internal_db:
  host: 172.15.0.2
  user: root
  passwd: dummypassword
  database: simulatorDB
```

```
simulator:
```

```
protocol: bb84
def_key_len: 128
chunk_size: 9
table: alice_sim_DB
chsh_threshold: -2.7
authentication: sphincs+

preshared_key:
  table: aliceSim_pre_keys
  presharedTTL: 20
  preshared_path: alicePresharedSimKeys

vault:
  host: 172.15.0.3:8200
  token: s.qSYNEKbCCQV1GE09QG4I0mwkd

channel:
  ip_addr: 172.15.0.6:4000
```

## Channel

Channel source code is located in `channel.py` file, under a folder named `channel`. Putting this code in another folder allows to manage its docker container through a dedicated Dockerfile, that is different from the one located in root folder (aimed to simulator instances). In this way, from `docker-compose.yaml` file, it is possible to build and execute target container with the command:

```
build: ./channel
```

Dockerfile will install all python requirements needed to execute the source code and will launch this implementation with the following command:

```
python channel.py
```

Even for channel it is possible to select some settings specified in a `yaml` file. This file allows to select among these settings:

```
attack:
  interceptAndResend: 0
  manInTheMiddle: 0

destinations:
  entangledSource: http://172.15.0.7:4000
  sourcePort: 4000
```

## Singlet source

Singlet source is the component in charge of producing singlets for E91 protocol. Source code for singlet source can be found in `singletSource.py`. This file is located in a folder named `singletSource` under root folder. This allows to specify a dedicated Dockerfile. This Dockerfile will be run from `docker-compose.yaml` file with the command:

```
build: ./singletSource
```

Dockerfile inside this folder will install python requirements and will launch singlet source application with the following command:

```
python singletSource.py
```

The peculiarity of this implementation lies on the attempt of speed up the singlet generation by prepare a given number of singlets before they are requested. The idea is to have singlets ready before each request, so that requests require the minimum possible time. In order to achieve such result, when source code is started, before creating the web interface a thread is spawned. This thread is in charge of constantly generating singlets until the default singlets number is reached. If from web interface more singlets than available are required, it is however possible to increase the number of singlets to be generated at runtime time. After each request, the number of currently available singlets is updated from web interface (depending on the requested number) and the thread starts over generating singlets to get ready for the next request. The default number of singlets to be generated is hardcoded in the source code and is defined at the beginning of the file as follows:

```
numberOfSinglets = 1024
```

Singlets are generated by let a Statevector object pass through the following qiskit circuit:

```
qre = QuantumRegister(2, name='qre')
cre = ClassicalRegister(2, name='cre')
singlet = QuantumCircuit(qre, cre, name='sgenerator')
singlet.x(qre[0])
singlet.x(qre[1])
singlet.h(qre[0])
singlet.cx(qre[0],qre[1])
```

### Simulator manager

Simulator manager source code can be found in `simulatorManager.py` file, located inside `manager` folder. This separated folder helps to manage its docker container with a dedicated Dockerfile. Hence, inside `docker-compose.yaml`, this container is set up with the following command:

```
build: ./manager
```

Dockerfile for this container is in charge of installing python requirements, unsealing vault storage and load preshared key. Unsealing is done through vault client application as already described in previous chapter. Preshared keys are loaded by running another python script called `load-PresharedKey.py` that interfaces to mysql and vault container to perform required actions. After that, Dockerfile launch the manager web interface with the following commands:

```
export FLASK_APP=run.py
flask run --host=0.0.0.0 --port=4000
```

Note that CLI interface is not launched in this instance. It should be launched by accessing container console as already explained in previous chapter. However, in order to access its console, container should be launched with interactive mode, with an access to its bash prompt. This is done in `docker-compose.yaml` file with the following configurations:

```
stdin_open: true
tty: true
```

### True random number generator

This implementation is located in python script `trng.py` in root folder. It is a module that can be directly imported where required and does not need a dedicated container. This module defines just one function that is in charge of simulate a true random number generation.

Defined function is pretty straightforward. Its name is `randomStringGen` and takes as argument an integer specifying the length of the random string to be generated. The generated string will be a sequence of '1' and '0' that can be interpreted as bits string and, when needed, converted in bytes to obtain an integer number. In order to simulate a true random number generation,

a Statevector object will be evolved through a quantum circuit containing Hadamard gates. In this way, qubits will be in superposition state and, when measured, will outcome with a random result. The quantum circuit employed in this implementation is the following:

```
q = QuantumRegister(temp_n, name='q')
c = ClassicalRegister(temp_n, name='c')
rs = QuantumCircuit(q, c, name='rs')
rs.h(range(temp_n))
```

## B.2 QKD Key Server and Module

Also this implementation runs over python 3.7 version. Source code can be found in *qkd-keyserver* folder. Inside this folder it is possible to find the key server source code, the docker-compose.yml file for automatic containers configuration and a folder named *qkd-module* that contains the QKD module implementation.

The docker compose file generates a subnetwork in which all containers operate. The subnetwork is defined as follows:

```
networks:
  frontend:
    ipam:
      config:
        - subnet: 172.17.0.0/24
```

Even for this implementation volumes folder has been used to specify different settings for the different containers. In particular, vault volumes for alice-vault container are the following:

```
- ./volumes/alice-vault/logs:/vault/logs
- ./volumes/alice-vault/file:/vault/file
- ./volumes/alice-vault/config:/vault/config
```

Where *./volumes/alice-vault* is the local folder contained in the root folder of the source code, while */vault/* is the destination folder inside the container in which selected folder will be mounted.

The same configuration can be found for bob-vault container, with the only difference that local folder used in this case is *./volumes/bob-vault*.

For mysql container a configuration similar to the previous case has been used. Here volumes property is specified as follows:

```
volumes:
  - ./db_init.sql:/data/application/db_init.sql
```

With *db\_init.sql* containing the database and tables used to initialise mysql instance.

The other volumes properties are used by the containers running the implemented stack composed by QKD key server and module. Even in this case the instances share the same source code, the only different parameters are located in yaml configuration files that are attached to each system exploiting volumes property. For alice-server container, this property is set as follows:

```
volumes:
  - ./volumes/alice:/usr/src/app/config
```

For bob-server instead, the current configuration is the following:

```
volumes:
  - ./volumes/bob:/usr/src/app/config
```

For all containers **on-failure** restart policy is specified, allowing the containers to restart their execution if an unexpected runtime error occurs.

## QKD Key Server

QKD key server is a python module running a Flask web server. A flask web server can be easily created and run with the following commands:

```
app = Flask(__name__)
app.run(host='0.0.0.0', port=serverPort)
```

Flask web server is used either from high level applications and key server peers. However, whilst communication between two key servers runs over http protocol, high level applications communicate with server by using https protocol. Hence key server uses a certificate in order to implement TLS protocol. Certificates used in this implementation have been generated by using OpenSSL command line application. In particular, a certification authority certificate has been generated with the following command:

```
openssl req -nodes -new -x509 -days 365 -keyout ca.key -out ca-crt.pem
```

After this step, a Certificate Signing Request (CSR) has been generated with the following command:

```
openssl req -nodes -new -keyout server.key -out server.csr
```

The CSR and the certification authority certificate have been used to generate a server certificate that will be used in TLS protocol:

```
openssl x509 -req -days 365 -in server.csr -CA ca-crt.pem -CAkey ca.key
-CACreateserial -out server.crt
```

The generated server certificate will be automatically used by flask web server when https protocol is in use, by defining a SSL context that will be passed to the web server interface:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS)
context.load_cert_chain('server.crt', 'server.key')
app.run(host='0.0.0.0', port=serverPort, ssl_context=context)
```

High level applications which need to talk with this server, will have to use the certification authority certificate generated here in order to successfully establish a TLS communication. This certificate needs to be explicitly indicated within the request since it is not a known certification authority. Using python 3.7, An example request of an high level application requesting status endpoint to key server is the following:

```
requests.get('https://172.17.0.6/api/v1/keys/targetSAE/status',
            verify='ca-crt.pem')
```

Note that the above request will authenticate the key server against the high level application. However, even high level application needs to be authenticated by the server and this is done through keycloak.

Keycloak allows to authenticate clients who require access to the key server implementation. Its working principle has already been described in previous chapter. Integration with flask web server is pretty straightforward. It is enough to add a configuration to flask object inside the source code as follows:

```
app.config.update({
    'SECRET_KEY': '\u0011\u00cf\u00fa\u00c\u00b9\u0095\u00e3\u00ba2\u007f\u00fd\u00ca
\u00a3\u009f\u0090\u0088\u00b8\u00ee\u00a4\u00d6\u00e4',
    'OIDC_CLIENT_SECRETS': 'client_secrets.json',
    'OIDC_VALID_ISSUERS':
        ['http://172.17.0.5:8080/auth/realms/quantum_auth'],
    'OIDC_RESOURCE_SERVER_ONLY': 'true'
})
```

Then it is enough to link the so configured flask object to an OpenIDConnect object:

```
oidc = OpenIDConnect(app)
```

With these simple operations it is possible to add the decorator on the methods that require authentication in order to achieve this functionality.

Keycloak configuration is composed by the following elements:

- `SECRET_KEY` is, as the name suggests, a secret key that can be self generated and is used during communication to keycloak server.
- `OIDC_CLIENT_SECRETS` is a link to a local file (available in the source code) that specifies client details. As previously said, key server as well must be registered as a client to keycloak realm in order to request tokens validation. When key server registers itself as a client, it gains a client configuration compose by an ID, a secret and other information. All this information is written in the `client_secret.json` file and will be used by the key server to authenticate against keycloak when request for information.
- `OIDC_VALID_ISSUERS` describes the realms allowed to release token for this implementation. If a valid token for another realm is used, the request will however be discarded since that realm is not a valid issuer for the key server.
- `OIDC_RESOURCE_SERVER_ONLY` specifies that this client will require only tokens validation and will not ask for an authentication token as high level applications do.

Keycloak and https are used to authenticate high level applications and key server. If a key server needs to talk with another one, none of these methods are employed. Authentication between two key servers is made through HMAC challenges, as already explained in chapter 10. HMAC is realised in python by using *poly1305-aes*<sup>1</sup> library. This library provides two simple methods, `authenticate` and `verify` which allow to generate and verify the message authentication code.

Key server can be configured through a yaml file. This file allows to select among the following settings:

```
global:
  preferred_qkd_protocol: BB84
  log_level: 0
  timeout: 300000

internal_db:
  host: 172.17.0.2
  user: root
  passwd: dummypassword
  database: alice_data

vault:
  host: 172.17.0.4
  port: 8200
  token: s.qSYNEkbCCQV1GE09QG4IOmwkd

settings:
  KME_ID: KME11223344
  ASSIGNED_ID_PREFIX: AABC
  DEF_KEY_SIZE: 32
  MAX_KEY_COUNT: 500
```

---

<sup>1</sup><https://cr.yp.to/mac.html>

```
MAX_KEY_PER_REQUEST: 3
MAX_KEY_SIZE: 32
MIN_KEY_SIZE: 8
MAX_SAE_ID_COUNT: 0
```

## QKD Module

QKD Module implementation can be found in QKDModule.py file. This file defines a class called QKDModule that includes the module interface and the web interface realised through flask. The peculiarity of this implementation lies on the key exchange mechanism. Once a key stream ID has been established between two QKDMs, they need to continuously exchange keys until a given number is reached. This key exchange is realised through a dedicated thread named QKDExchange that is started as soon as OPEN\_CONNECT function is called on the both of the QKDM instances.

Another implementation details that is worth to be described is the registering phase of the QKDM to the key server. This step is realised through the southbound interface function QKD\_REGISTER\_MODULE. When QKDM calls this function, key server returns the access to vault storage and mysql database. Besides the IP addresses needed to access the storage containers, key server provides the token to authenticate against vault and a dedicated path where only this QKDM can write and read secrets. Python library for vault is named *hvac*. An example of reading secrets from vault by using this library is the following:

```
client = hvac.Client(url='http://172.17.0.4:8200')
client.token = 's.qSYNEKbCQV1GE09QG4I0mwkd'
response = client.secrets.kv.read_secret_version(path='storedkeys')
keys = response['data']['data']['keys']
```

For QKD module a dedicated configuration file has been defined. Configuration file allows to select if the underlying simulator should use BB84 protocol or a dummy protocol that generates random keys without waiting for a real key exchange. This file is defined as follows:

```
simulator:
  protocol: bb84
  # protocol: dummy
```